Learning with your computer

$4.95
NZ $6.95 Inc. GST

ACHIEVE MORE WITH YOUR PC

HyperCard — how to use it!

ProDOS Tutorial

Hints and Tips for Turbo Pascal

Batch File Encyclopedia

HEURISTICS — problem solving with a computer

A guide to NETWORKING

Design your own Animation and CAD systems

THE Lifestyle SERIES

The Lifestyle Series offers you information on ways to improve your home-design a new kitchen or bathroom, plan your outdoor living area, renovate or redecorate your favourite rooms.

- Restorations & Renovations
- Design & Decorating
- Pools & Spas
- Kitchens
- Pools & Outdoor Living
- Bathrooms
- Home Improvements.

OUT NOW!

...AN INNOVATIVE NEW SERIES OF SPECIALIST MAGAZINES FOR TODAY'S HOMEMAKERS

Learning with your computer

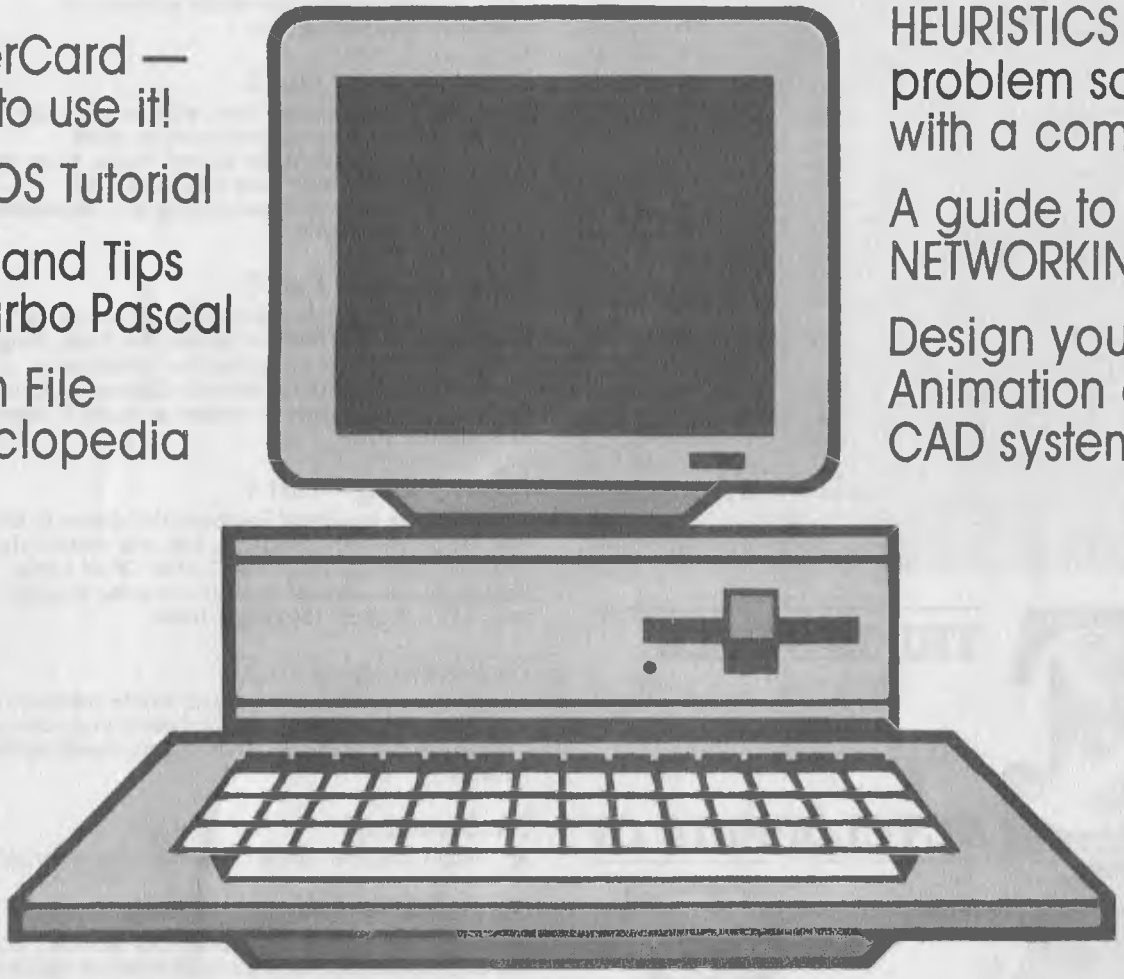
ACHIEVE MORE WITH YOUR PC

HyperCard — how to use it!

ProDOS Tutorial

Hints and Tips for Turbo Pascal

Batch File Encyclopedia



HEURISTICS — problem solving with a computer

A guide to NETWORKING

Design your own Animation and CAD systems

Welcome to Learning with Your Computer!

Every month in *Your Computer* magazine we present articles and features designed to teach users how to gain from a Personal Computer – after all, it's a productivity tool; if it doesn't help you achieve more, then it's a waste of time. 'Achieving more' with your PC is what these pages are about – if you haven't already realised it, a computer is a powerful device that can be taught to do exactly what you want it to.

If you've got a taste for artificial intelligence, try the simple problem solving program in 'Heuristics' – it gives a good indication of just how smart a dumb machine can be. A step up from that is 'Graphics Techniques' which shows how to design your own simple animation system and gives enough background to enable you to build a complex, animated display; the series also demonstrates how to program a simple Computer Aided Design system. One of the most powerful and straightforward ways of 'customising' a PC to do what you want, is with batch files and one of the most powerful Batch Filers around is Les Bell: his 'Encyclopedia' offers a wealth of tips for these useful tools. HyperCard is a revolutionary concept for Apple's Macintosh, but it has implications for the IBM world too – already we're seeing 'hypermedia' products for DOS machines that are changing the way we think when working with software. And, if you're curious about 'connectivity', Stewart Fist's 5-part series, 'Networking', puts the whole field in perspective, sorts out the jargon and demonstrates that one computer plus another is much more than two.

***Learning with Your Computer* is the place to start learning more about *your* computer!**

CONTENTS

Cover photograph: Don Carroll/The Image Bank

Editor
Jake Kennedy

Production Editor
Allecia Khartu

Cover Design
Pamela Horsnell

Production
Kylie Prats

Managing Editor
Michelle Smith

Publisher
Michael Hannan

EDITORIAL
AND OFFICE SERVICES
Allison Tait
180 Bourke Rd,
Alexandria 2015 NSW
Tel: (02) 693 6620
Fax: (02) 693 9935

ADVERTISING SALES OFFICES

New South Wales
Sales Executive
Mark Wilde
180 Bourke Rd,
Alexandria 2015
Tel: (02) 693 6666
Fax: (02) 693 9935

Advertising Production
Patrice Wohlnick

Victoria
Sales Manager
Anne Willey
3rd floor, 615 St Kilda Rd, Melbourne
Tel: (031 525 1010,
Fax: (03) 529 2997

Advertising Production
Matt Holden
221A Bay St, Pt Melbourne 3207
Tel: (03) 646 3111,
Fax: (03) 646 5494

Western Australia
Des McDonald
48 Clieveden St, North Perth 6006.
(09) 444 4426
Fax: (09) 381 3115

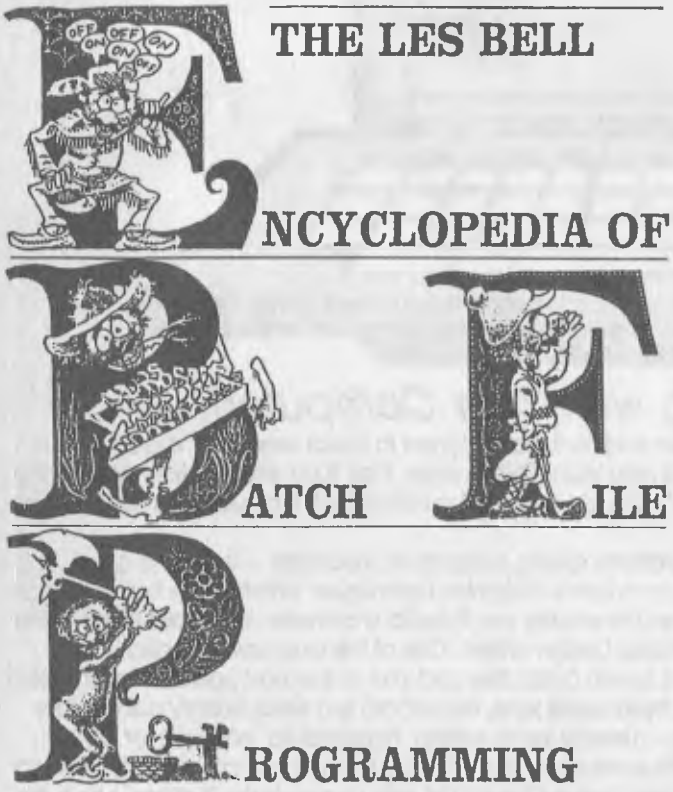
Queensland
John Saunders
26 Chermside St, Newstead 4006
Tel: (07) 854 1119

South Australia
Michael Mullins
98 Jervois Street, Torrensville 5031.
Tel: (08) 352 7937

New Zealand
Corrie Mitchell
Rugby Press
3rd Floor, Communications House,
Parnell, Auckland.
Tel: (09) 796 648
Tlx: NZ 63112 SPORTBY
(02) 693 9517

LEARNING WITH YOUR COMPUTER 1989
was published by
The Federal Publishing Co Pty Ltd,
180 Bourke Rd,
Alexandria 2015 NSW.
Printed by HannanPrint,
140 Bourke Rd, Alexandria 2015.
Distributed by
Newsagents Direct
Distribution Pty Ltd.

*Recommended and maximum price only.



CONTENTS

HEURISTICS

Rules of thumb for problem solving

Computers need black and white choices to make a decision — so how can they be taught to make a choice when the problem is more complex than that? Easy, says Tim Hartnell . . .

HEURISTICS ARE 'rules of thumb'. They are often rather inexact rules, which have been proved in practice to produce correct results, more often than not. Heuristics are important in artificial intelligence (AI). As they are, in effect, paths or signposts towards solutions, an AI program can employ them to solve problems. But computers don't like things which are not either black or white, zero or one. Shades of grey, degrees of uncertainty, and possibilities and perhaps sit unhappily within electronic brains.

Nevertheless, the intuitive judgement, the exercise of common sense, is a vital part of human thinking, and it is said that until an AI program can work with incomplete data, and imperfect methods, and still get its results, it has no reason to expect to be considered intelligent.

'Heuristics are criteria, methods, or principles for deciding which among several courses of action promises to be the most effective in order to achieve some goal.' That's the definition given by Judea Pearl in *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, (Addison-Wesley, 1984). These heuristics, says Pearl, 'represent compromises between two requirements: the need to make such criteria simple and, at the same time, the desire to see them discriminate correctly between good and bad choices.'

All problem-solving programs have procedures with which to work. I suggest that only if the procedure is imperfect, unproved or inexact can it really be considered to be a heuristic procedure. In an effort to make that statement a little more clear, I've included a program with this article which shows a heuristic approach in action.

The program, Antimind, is really Mastermind in reverse. Instead of the computer thinking of a four-digit code which you have to solve, the program tries to solve your three-digit code (in which none of the numbers appears more than once). Think about the last time you played Mastermind. How did you narrow in on the correct choice? It is extremely difficult to determine exactly how you made your decisions, apart from the obvious one of rejecting all colours in a choice if you've no blacks or whites for that guess.

Antimind is an attempt to write a program which would work towards the answer to a problem in an apparently intelligent manner, even when no clearly-defined human approach to solving the problem exists. In this, the computer is attempting to solve a Mastermind-like puzzle, in which you think of a three-digit numerical code, and the computer tries to guess it.

You give the computer feedback in the form of responding to its guesses with 'whites' and 'blacks', where a white is given for a digit which is correct, but is in the wrong position within the code, and a black is given whenever there is a correct digit within the code.

The problem, as I suggested earlier, is not totally straightforward, as the computer does not know, for certain, which digit produced which result. It works in a fairly simple manner, in theory, although implementing the relatively simple idea behind the program was not particularly easy. Every time a digit appears in the code which is awarded a black, every digit within that code is weighted so it appears more often in future guesses. The more blacks in that particular code, the higher the weighting each code gets. A much smaller weighting is awarded if the code gets one or more whites. Any code getting neither a black nor a white leads to all the digits within that guess being totally removed from future consideration.

The program works reasonably well, in most cases, although it can sometimes get a fixation for a digit which is not in the answer, and which it will then bring up endlessly, refusing to discard the wrong answer. Once you're tried it a few times, you might want to refine its methods, or try a totally different approach to solving the problem (and I'd like to see any solutions you come up with).

Here is the output for one run of the program, when it managed to solve it within seven guesses —

```
Guess number 1
My guess is 1 2 3
How many blacks? 1
And how many whites? 0

Guess number 2
My guess is 4 5 6
How many blacks? 0
And how many whites? 1

Guess number 3
My guess is 7 8 9
How many blacks? 1
And how many whites? 0

Guess number 4
My guess is 4 2 6
How many blacks?
And how many whites? 1

Guess number 5
My guess is 8 2 1
How many blacks? 0
And how many whites? 1

Guess number 6
My guess is 1 9 4
How many blacks? 1
And how many whites? 2

Guess number 7
My guess is 1 4 9
How many blacks? 3

I guessed your code of 1 4 9
in just 7 guesses
```

If you want to try your hand at other problems of this type, you could write a program to solve the following puzzles:

Puzzle One: There are twelve billiard balls, eleven of which are identical in weight. The remaining ball — the odd one — has a different weight. You are not told if it is heavier or lighter. You have a balance scale for weighing the balls. You

```
10 REM ANTIFOUR - to solve four digit codes
20 REM (c) Tim Hartnell, 1987
30 GOSUB 520:REM INITIALISE
40 REM MAKE A GUESS
50 IF GUESS = 0 THEN FOR Z = 1 TO 4:B(Z) = Z:NEXT Z:GOTO 80
60 IF GUESS = 1 THEN FOR Z = 1 TO 4:B(Z) = Z + 4:NEXT Z:GOTO 80
70 GOSUB 290
80 CLS
90 GUESS = GUESS + 1
100 PRINT:PRINT
110 PRINT "Guess number"GUESS
120 PRINT
130 PRINT "My guess is"B(1);B(2);B(3);B(4)
140 PRINT:PRINT
150 INPUT "How many blacks";B
160 IF B = 4 THEN GOTO 590
170 IF B = 3 THEN 200
180 PRINT:PRINT
190 INPUT "And how many whites";W
200 IF W + B = 4 THEN Q = 4:C(1) = B(1):C(2) = B(2):C(3) =
    B(3):C(4) = B(4)
210 IF B + W = 0 THEN C(B(1)) = 0:C(B(2)) = 0:C(B(3)) = 0:C(B(4))
    = 0:GOTO 40
220 IF B > AID THEN FOR Z = 1 TO 4:E(Z) = B(Z):NEXT Z:AID = B
230 FOR Z = 1 TO 9
240 FOR D = 1 TO 4
250 IF B(D) = C(Z) THEN C(Z) = C(Z) + (B + W)*100 + W*10
260 NEXT D
270 NEXT Z
280 GOTO 40
290 REM Pick four numbers
300 FOR Z = 1 TO 4
310 D1 = C(INT(RND*Q) + 1)
320 IF D1 = 0 THEN 310
330 D2 = C(INT(RND*Q) + 1)
340 IF D2 = 0 THEN 330
350 IF INT(D1/10) > INT(D2/10) THEN B(Z) = D1
360 IF INT(D1/10) < INT(D2/10) THEN B(Z) = D2
370 IF INT(D1/10) = INT(D2/10) THEN B(Z) = D1
380 IF B(Z) > 100 THEN B(Z) = B(Z) - 100*INT(B(Z)/100):GOTO 380
390 IF B(Z) > 10 THEN B(Z) = B(Z) - 10*INT(B(Z)/10):GOTO 390
400 NEXT Z
410 IF B(1) = B(2) OR B(1) = B(3) OR B(1) = B(4) OR B(2) = B(3)
    OR B(2) = B(4) OR B(3) = B(4) THEN 300
420 IF AID > 0 THEN COUNT = 0:FOR Z = 1 TO 4:IF B(Z) = E(Z) THEN
    COUNT = COUNT + 1
430 IF AID > 0 THEN NEXT Z:IF COUNT < AID - 1  THEN 300
440 M = 1000*B(1) + 100*B(2) + 10*B(3) + B(4)
450 K(GUESS) = M
460 IF GUESS < 3 THEN 510
470 COUNT = 1
480 COUNT = COUNT + 1
490 IF K(COUNT) = M THEN  300
500 IF COUNT < GUESS - 1 THEN 480
510 RETURN
520 REM INITIALISE
530 GUESS = 0:Q = 9:AID = 0
540 DIM B(4),C(9),E(4),K(100)
550 FOR Z = 1 TO 9
560 C(Z) = Z
570 NEXT Z
580 RETURN
590 PRINT:PRINT
600 PRINT "I guessed your code of"B(1);B(2);B(3);B(4)
610 PRINT TAB(5);"in just"GUESS"guesses"
620 END
```

Listing 1. *Antimind — an attempt to write a program which would work towards the answer to a problem in an apparently intelligent manner, even when no clearly-defined human approach to solving the problem exists.*

'Heuristics are criteria, methods, or principles for deciding which among several courses of action promises to be the most effective in order to achieve some goal.'

have to find out which ball is the odd one — using only three weighings — and whether it is lighter or heavier than the others.

Puzzle Two: There is a checkerboard which has had it's upper left and lower right squares removed. You have a box of dominoes which are one square by two squares in size. Can you exactly cover the checkerboard with dominoes?

Puzzle Three: There are four people: Roberta, Thelma, Steve and Pete. Among them, they hold eight different jobs (no wonder the dole queues are so long, with these four hogging the jobs). Each holds exactly two jobs. The jobs are chef, guard, nurse, telephone operator, police officer, teacher, actor and boxer. The job of nurse is held by a male, and the husband of the chef is a telephone operator. Roberta is not a boxer, and Pete has no education past the ninth grade. Roberta, the chef and the police officer went golfing.

According to *Automated Reasoning: Introduction and Applications* by Larry Wos, Ross Overbeek, Ewing Lusk and Jim Boyle (Prentice-Hall, 1984), the source of these problems, the final one regarding the jobs has been solved by intelligent sixth-graders, so your computer should be able to handle it.

I'd be interested in seeing programs which solve these, and similar problems. If you don't want to type in the Antimind program, you can download it from our Bulletin Board, along with an extended version of the program, Antifour, which solves four-digit codes. Alternatively, I can supply you with both programs, along with a number of other interesting programs, on a disk for the IBM PC for $5.00. You can send your programs and comments to me at this address: Tim Hartnell, 34 Camp Street, Chelsea, Vic., 3196. □

COMING TO GRIPS WITH NETWORKING

- Part 1

Stewart Fist introduces the concepts behind networking by explaining why, when you talk about LANs, you encompass the whole universe of computer communications.

I SUPPOSE THE most primitive form of Local Area Network (LAN) is the cable that connects your computer to your printer. It's a network of sorts, even though it only provides a link to a peripheral. For a number of years I have had two PCs linked to the same daisy wheel printer. There's just one length of cable with an extra plug soldered onto the middle. I could have gone to all the trouble to put switches in the system, but everything works fine as long as both computers don't attempt to print at the same time.

You may reject the claim that a cable link between one computer and a printer is a LAN, but what about two? My double-link to the printer is a three 'node' (two computers, one peripheral) network without any collision avoidance scheme — if you don't count the yell: 'Are you using the printer?' before we hit return.

The point is that there are LANs and there are LANs. At one end of the spectrum we have simple cable systems for peripheral sharing at distances of only a couple of metres and at the other we have complex broadband communications networks with voice, video, and data streaming down cables and over micro-wave links spanning dozens of kilometres. These days when you talk about LANs, you are almost encompassing the whole universe of computer communications.

So we need to create our own definition. If we reject the PC-printer connections as being too simple, there are probably three criteria that define a LAN:

1) The devices on the network share information along common pathways.

2) A method exists which prevents data from one user being corrupted by data from another. If they share common pathways, then it is essential that the system has some way to avoid data collision and corruption.

These days when you talk about LANs, you are almost encompassing the whole universe of computer communications.

3) The devices are addressable — if I need to send data to a hard disk unit, it won't also simultaneously appear on the printer, or on the screens of other computer users in the system.

As a general principle, to satisfy these requirements the data needs to be transmitted along the common pathway like a train with numerous characters in a chain. These trains of data are called 'packets' or 'frames' and each frame will need to contain a header with the address information (where is it going to and who is it from?), the position of this packet in the sequence (is this the third or fourth frame?), then the data, followed by some form of checking sequence to make sure everything sent has arrived.

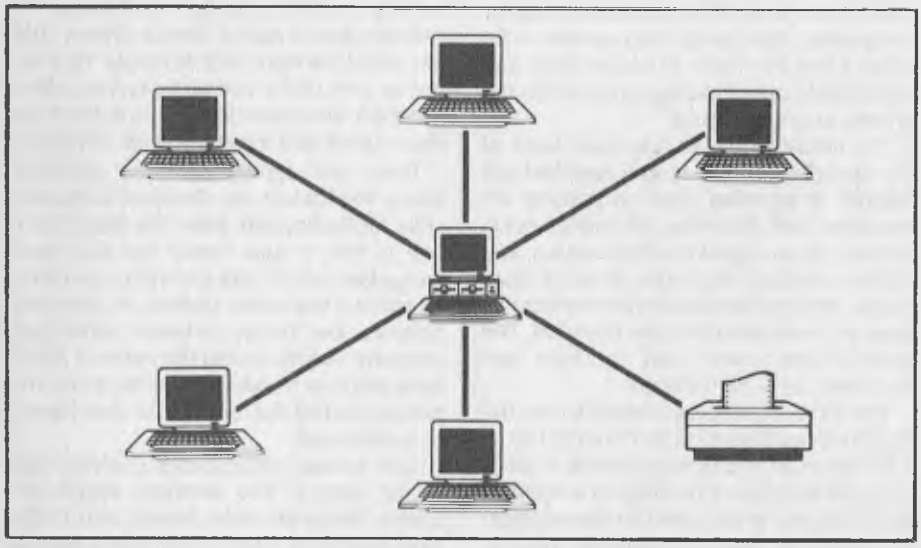
AppleTalk

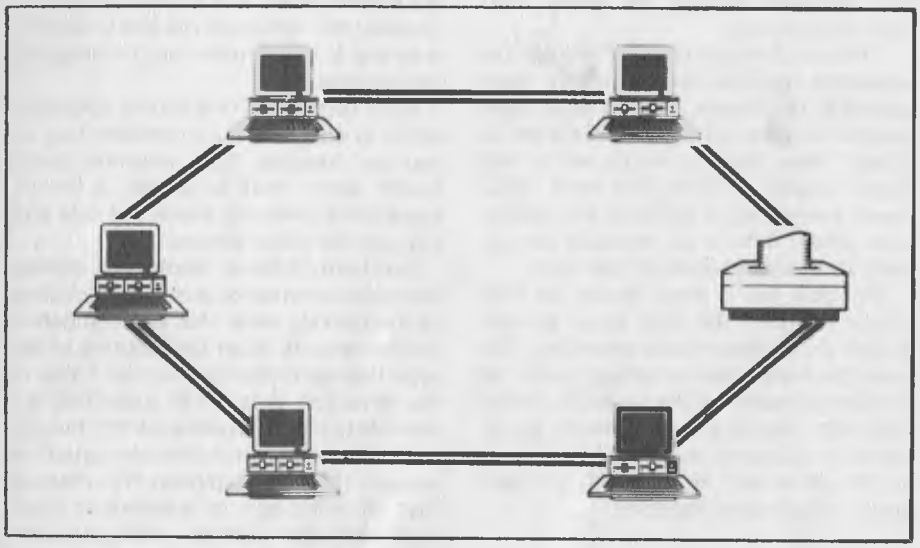
AppleTalk is pretty well the lowest form of network life that we can classify as a member of the LAN family if we accept the above as adequate. (Although I'll be discussing Apple and AppleTalk in particular in this article, the principles generally remain the same regardless of the system.) And despite some early criticism that Apple didn't have a file-server (it does now), AppleTalk has proved to be an excellent work-group solution to the problem of peripheral and resource sharing.

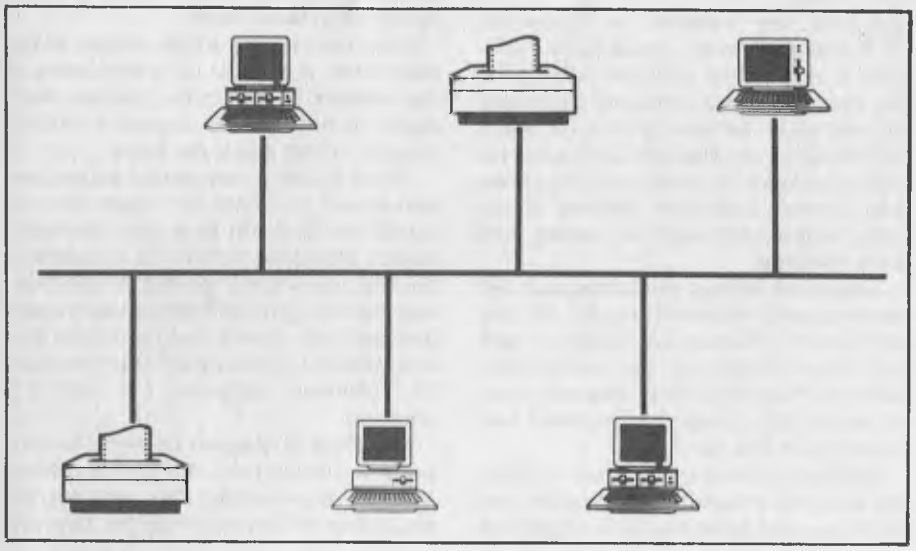
It is a low-to-medium speed LAN with a raw-data transmission rate of only 230.4K bits per second and it can handle only 32 nodes over a distance of 300 metres, maximum. So it's certainly not the Superman of the network world, but it is a good starting point for looking at LANs.

First of all we need to look at topography — an erudite way of saying 'what is the basic design?' There are three choices (plus a number of combinations and sub-choices). We can link everything together in one line; we can join the ends of this together to form a ring; or we can radiate all our links out from some central hub. Respectively, these three are the Linear Bus, Ring or Star topographies.

AppleTalk is a linear bus design which uses multi-drop lines linking the bus nodes to each computer or peripheral. Surprisingly little hardware is involved and the system has a lot of flexibility.







The Hardware

The 'network' itself is a series of shielded, twisted-pair cables with an impedance of 79 ohms. You can buy made-up cables with three pin DIN connectors attached in lengths of two and ten metres from Apple, or you can make your own. The cables are used to provide the links between 'connection modules' which give a passive junction between each 'node' (device) and the main trunk cable. This passivity is important because it means that a node can be added or removed with only a minor disruption to the service. If any one part of the system fails, it probably won't disturb the overall functioning.

The connection module is a plastic box the size of a cigarette packet which surrounds a transformer and a few resistors and capacitors. A 45 cm cable links this box to the 'node' (computer, printer, and so on) while two DIN sockets on the sides provide links from and to the next module in the chain. The two pins in one DIN connector are electrically continuous with the two pins in the other whenever a cable is plugged in.

The transformer isolates the node-link from the main trunk while a resistor provides an automatically switched terminat-

Star Network: *Star designs (top) need a network server (usually a dedicated computer at the 'hub' of the star). If the network server crashes, the whole network goes with it; however, if a peripheral fails, the system continues uninterrupted. Over distances up to 600 metres, star networks are usually connected by twisted-pair cables (as are telephones, incidentally); for longer distances, coaxial cable or optical fibre can be used.*

Ring Network: *The ring (center), also known as a daisy chain, is a large electronic traffic circle. All PCs and peripherals are connected together in a circle and all information passes through each node in the system. There is usually a parallel path around each node which provides a detour if a peripheral crashes. The nodes can be connected to twisted-pairs, coaxial cables or optical fibre.*

Linear Bus Network: *A bus structure (bottom) usually uses a main trunk twisted-pair or multi-wired cable into which the peripherals and PCs are connected. One of the advantages of bus topography is that a wide range of equipment can be hung on the LAN without problem. Dissimilar equipment — Macs, IBM PCs, Apple IIs, LaserWriters, and different modems — may not be able to talk to each other but they can share the use of the cable and compatible peripherals.*

ing load across the line if a second extension cable is not plugged into the module; for example, if it is the end of the line. This is needed to stop data pulses 'echoing' back down the line from the open end — like an organ pipe.

All the other capacitors and resistors increase the noise immunity of the nodes and provide protection from possible ground-loop currents. The result is an electrically balanced, transformer-isolated, serial-communications, multi-drop chain which, for all the jargon, is very cheap and easy to build.

AppleTalk can handle up to 32 nodes per network, but networks can be interlinked by bridges and translators so the range of possible connections on one 'internet' is very large — more about bridges and translators later.

You simply plug in the connection modules, link them with shielded cables, plug in your software, and go!

If you've got a number of Apple Macintoshs in an office, the above is more than you will ever need to know about the hardware. AppleTalk is relatively foolproof and anyone can set up a system. You simply plug in the connection modules, link them with shielded cables, plug in your software, and go!

The Mac is able to provide this simplicity because it has a lot of network electronics already built in. You can connect Apple IIs and IBM PCs into an AppleTalk network but you need a special communications board/serial interface for each.

AppleTalk is based around the use of Zilog's 8530 programmable communications chip; it is a highly flexible device that can be programmed to use a variety of communications methods and protocols including RS-232C and RS-422 (virtually a balanced version of RS-232C). AppleTalk uses RS-422 because balanced lines are less likely to suffer interference over distance.

There are two more chips that provide support for the RS-422; a driver chip (26LS30) and a receiver chip (26LS32) which are both low-cost and readily available. In operation the receiver chip is always listening on the line and it will pass on any data to the Zilog controller chip for processing. The driver chip is only activated when the node is transmitting and ideally only one driver is connected to the system at any one time.

The entire hardware (physical) layer of the AppleTalk system is well specified but flexible. It is called upon to perform bit encoding and decoding, bit transmission and reception, signal synchronisation, and carrier sensing; the entire physical layer can be replaced by some other medium as long as these functions are provided. The specifications aren't tied to chips and hardware, only to functions.

The 8530 chip is programmed to use the Synchronous Data-Link Control (SDLC) as a bit-oriented line protocol which establishes the way data transmission is regulated. SDLC was widely used for communication with IBM mainframes and so this control capability is built into many communications chips.

The main function of the 8530 chip is to assemble the data and retrieve it from 'packets' or 'frames' which have been marked by special bit-patterns known as 'flags'. These frames can be up to 600 bytes (characters) long, but since SDLC treats everything in terms of bits (rather than bytes) there is no necessity to conform to the standard 8-bit byte size.

The chip has a mode known as FM0 which encodes the bits by a process known as bi-phase space encoding. This uses the transitions in voltage levels on the line to represent the 1s and 0s, rather than the absolute voltage levels themselves — a system somewhat analogous to the phase-shift keying (PSK) principle used in high-speed modems.

Software

While the hardware is reasonably straightforward, AppleTalk's software is much more complex, although it has been compressed into only 6 kilobytes of code which becomes part of the Mac's operating system. The software has to provide translation of names and functions into network addresses, delivery of the data to the right address, routing, and error checking.

AppleTalk follows the conceptual ISO seven-layered reference model for the connection of computer systems, with each layer building on the services provided by those beneath it. However, most of AppleTalk's layers are simplified versions of the OSI standards.

Protocols in the top five layers — physical, data link, network, transportation, and session — are at the core of the AppleTalk software and these are included in a set of routines and a pair of device drivers that are called the AppleTalk Manager. This becomes part of the operating system when installed and provides the link between the network and the application program.

There are several different software layers. The Link Access Protocol is responsible for finding out when the main trunk line is free. It also sends out data and recognises which data should be received by reading the frame address. It therefore provides the bridge between individual computer functions and the network hardware and flow of data, so it is the most important part of the network for developers to understand.

Link Access Protocol uses CSMA/CA for access control. The acronym stands for Carrier Sense Multiple Access with Collision Avoidance. The Carrier Sense part of the phrase means that a node wanting to transmit first listens on the line to see if it is in use. If it is, it defers to the on-going transmission.

It's a first-come, first-served operation which is okay for small networks but, as you can imagine, large networks using Carrier Sense could be locked up by one computer transferring masses of data and hogging the whole system.

The term Collision Avoidance defines the technique used to avoid data colliding in the unlikely event that two computers on the network begin transmitting at exactly the same time (within the limits of the detection delay). On AppleTalk, all transmitters wait listening on the line for a short period of time after any data flow ceases. Then, to minimise the chances that all will begin to transmit at once, each computer adds an additional random delay which varies according to the perceived network traffic.

If by chance two stations start up at the same time, it is up to the other layers in the network to detect the garbled messages on the line and request a retransmission of that particular frame.

There is also a very similar access system known as CSMA/CD, where the CD stands for Collision Detection. Here the station listens on the line as it begins to transmit and if it detects that its own message is being garbled, it stops, waits a random amount of time, and then if the line is still clear it begins again. There are subtle difference between CA and CD schemes.

AppleTalk's Datagram Delivery Protocol works in conjunction with Link access by handling data routing. This includes the forwarding of data over 'bridges' between

various networks. To do this it must work in conjunction with the Routing Table Maintenance Protocol which each bridge uses to keep track of all other networks available to fit it and also which bridges provide the shortest pathway to the destination.

Your applications would link to AppleTalk through software entry points known as 'sockets' and the system provides two alternative protocols for supporting these sockets. If you are involved in question and answer type sessions (transaction-oriented use), the Transaction Protocol manages the message sequencing and timing of the request/response format.

Long streams of data — sending or receiving — are handled by the Data Stream Protocol which changes the streams into packets and sends them on their way. It also checks received packets for errors, demands retransmission if errors are detected, and makes sure that duplicate packets are removed. The integrity of each packet is ensured by the use of a 2 byte (16-bit CRC) checksum at the end of each frame.

There are a number of other protocols: a Transaction Protocol which handles requests and responses between applications and the 'sockets', a Name Binding Protocol which allows you to use your own names to indicate network nodes and Filing and Printer Access Protocols which handle the file-servers and the printers, including any LaserWriters.

AppleTalk uses a 1 byte (8-bit) addressing number which is attached to the head of each frame of data. This node identification can therefore be any number up to 255, although 255 itself is reserved for broadcast use — everyone on the system will read it.

The node IDs aren't fixed, but are allocated at random by the system software itself. When a machine first comes on line it generates a random number as a trial ID. It then butts into the network and sends a dummy message to this ID. If it doesn't get acknowledgement it knows that this number is free and it will continue to use it for the duration of the on-line session. If not, it will try again.

In a way this typifies the whole approach that the AppleTalk designers have taken. It's not an elegant solution to the problems of identification but it does work and it doesn't require the time and attention of a network specialist — and this is a philosophy that Apple strictly adhere to

In the next part of the series we'll be looking deeper into networking. Until then, keep a grip on the LAN situation. □

Coming to grips with Networking - Part 2

Local area networks have been with us since the mid-70s when they were developed to allow expensive peripherals to be shared. Today, there are dozens of systems that have built up a loyal following, but, if there is such a thing as a networking standard today, it is Ethernet.

LOCAL AREA Networks (LANs) are not new; they've been around, albeit in primitive form, since the mid 1970s when their main purpose was to link central processing units to expensive hard disk drives and printers. In the '70s, the costs of computer processing were plummetting, but the price of peripherals remained high. You could buy a complete computer for less than the price of a printer, so LANs were introduced as a method of cost saving.

Corvus Systems was the first in the microcomputer LANs market with its Omninet, designed primarily to allow microcomputers to share one of Corvus's hard disk drives. At the time, it was *the* major manufacturer of hard disks and therefore had a vested interest in getting these multi-thousand dollar items linked to microcomputers.

Corvus is still around with Omninet — mainly in the educational and small-business area — and has a very large installed base of LANs systems, together with a rather unenviable reputation for unreliability. But Corvus are still improving their system by widening the range of computers that can link in, and the operating systems they can handle, and designing new bridges to mainframes and to other networks.

There are dozens of other small LAN systems that over the years have built up a loyal band of followers — and an equal number of highly vocal critics. There's ArcNet, IANET, Fox 10-Net, Sytek, DR-Net and PC-Net II, to name but a few.

Ethernet

But today, the term LANs very largely equates with Ethernet. If there is such a thing as an industry-wide standard in networking, Ethernet is it (although the IBM Token Ring system is starting to give Ethernet a run for its money).

Ethernet was developed by Robert Metcalfe at Xerox. Metcalfe went on to found 3Com, a company which now specialises in highly standardised Ethernet network systems. Xerox joined together with Digital (DEC) and Intel, the chip-maker, to force Ethernet on the industry as a standard — and much to the surprise of everyone, they have largely succeeded.

The reasons seem to be that Ethernet proved to be moderately fast, quite reliable, very flexible and capable of taking on new roles. To top this praise off, Ethernet seems to now be implementable on a wide range of different media — although it was designed specifically for coaxial cable connection.

If there is such a thing as an industry-wide standard in networking, Ethernet is it . . .

It was originally specified as a 10 megabits/second baseband bus system and this remains the basic world-wide standard, although there is a star-topology version of Ethernet around. Ethernet systems have been implemented on everything from twisted-wire pairs to optical fibre and over both microwave and infrared aerial links. Some systems even incorporate Ethernet as a channel in broadband networks where it happily co-exists with the normal broadband LANs.

Recently, a French firm has managed to double the capacity of Ethernet while still retaining the 10 megabits/second standard, and it also claims to have introduced a voice along with the data-stream — something that was only thought possible using broadband techniques.

So it came to pass that in August 1983, the IEEE announced the 802.3 standard with specifications close enough to those of Ethernet to constitute a tacit endorsement of its physical and control protocols.

CSMA/CD

Ethernet uses a Carrier Sense, Multiple Access/Collision Detection (CSMA/CD) control protocol which is derived from a system called Aloha, developed — you might have guessed — at the University of Hawaii. I quickly described the Ethernet CSMA/CD collision avoidance system in Part 1 when discussing AppleTalk, but it is well worth looking at in more depth.

This is a first-come, first-served protocol which works fine up to about 80 per cent of the maximum theoretical capacity of a system; above this, the system gets bogged down. When too many computers attempt to access the common cable at the same time, the whole system starts spending too much time contesting who owns the communications path.

From the time that it was released in 1979/80, Ethernet was involved in a public brawl with the International Standards Organisation (ISO). Ethernet didn't conform to the organisation's Open Systems Interface (OSI) seven-layer connection standard and for a while it appeared that Ethernet was doomed.

It is a measure of the quality of the original design that the American Institute of Electrical and Electronic Engineers (IEEE) came to its rescue by setting a committee which had the job of devising specific standards for LANs interconnection which allowed for Ethernet's CSMA/CD implementation.

At about the same time the European Computer Manufacturer's Association adopted Ethernet as its LAN standard and a number of major mainframe computer manufacturers in the USA followed suit. So it came to pass that in August 1983, the IEEE announced the 802.3 standard with specifications close enough to those of Ethernet to constitute a tacit endorsement of its physical and control protocols.

Since that time Xerox and its partners have further defined Ethernet for most of the other ISO levels — although the US Department of Defence has managed to independently define its own conflicting standards for levels three and four.

Basically it works as follows — when a node wishes to send a message over the network, it first listens on the LAN for activity and if the network is silent it begins transmitting. However, if it finds that the network is busy, it will wait — constantly listening for the conversation to end. When it does, the waiting node will immediately try to seize control and transmit its message.

Most data collisions occur at this point because 2 or more nodes might be waiting and will all try to transmit at the same time. This results in a jumble of data, none of which makes sense to anybody listening and since each Ethernet controller also has a receiver that simultaneously monitors all communications on the line, the controller will immediately detect the overlapping pulses caused by the collision and stop transmitting.

All competing nodes will react the same and all will immediately transmit an abort pattern over the LANs informing everyone on the network that a collision has occurred and therefore that the last stream of data was garbled. The receiving station/s immediately dump their registers of data received and the whole network goes into a holding pattern.

Each node waiting to transmit then loads itself with a random number that it uses to represent a time period — and it waits this amount of random time before it begins to retransmit. The first node to time-out therefore has a high chance of finding the network free, and once it begins to transmit the whole wait-cycle for the other nodes begins again.

This process of network contention works very well in Ethernet for up to a 1000 or so nodes in normal circumstances but obviously the greater the number of nodes sharing the network, the greater the chance of collisions.

The information in an Ethernet network is transmitted and received in packets, each of which is called a 'frame'. These frames consist of: a Preamble, two address fields, a Type field, the Data field and a frame check sequence, in that order — refer to Figure 1.

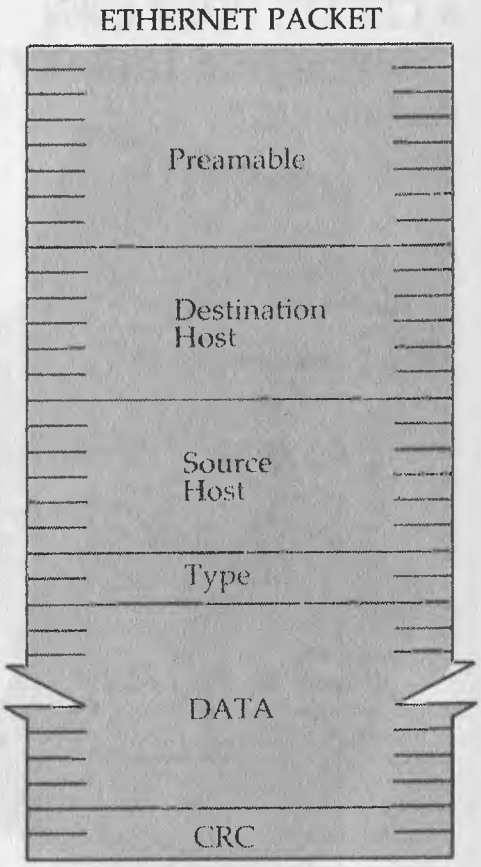


Figure 1. The information in an Ethernet network is transmitted and received in Packets, each of which is called a 'frame'. These frames consist of: a 64-bit opening statement, the Preamble; a 6-byte Destination address; a 6-byte Origination address; a 2-byte Type field that defines the user protocol associated with the frame; a Data field, which must be longer than 45 bytes and shorter than 1500; and a frame Check sequence, which ensures the data received is the same as that sent.

The Preamble is a 64-bit opening statement that synchronises the receiver and the transmitter and generally advises everyone on the network that data is on its way. It consists of 62 alternating ones and zeros, followed by an end-mark pair of ones.

The Destination address follows immediately after and it consists of a 6-byte

BINDER OFFER!

Can't Find It? File It!

These attractive, ready to use, sky blue vinyl binders have been specially designed to hold and protect 12 of your valuable magazine collection in the easy clip-in fastener wires.

★ CREATE YOUR OWN REFERENCE LIBRARY ★

10% DISCOUNT FOR ORDERS OF SIX OR MORE BINDERS

A must for regular readers.

Please send me ..... binders @ $8.00 each = $ .....
PLUS postage and handling @ $2.00 each = $ .....

For ..... TOTAL $ .....
(Magazine Name)

☐ I enclose my cheque/money order (with this form in an envelope) for $ .....

Charge by ☐ Bankcard ☐ Mastercard
☐ American Express ☐ Visa with $ .....

Card No. Expiry Date / /

Signature .....
(unsigned orders cannot be accepted)

Mail Post Free in Australia to:

Freepost No. 4
The Federal Publishing Company
P.O. Box 227
Waterloo, N.S.W. 2017

Name: ..... *Mr/Mrs/Ms/Miss* *Initial* *Surname*

Address: .....

..... *Post Code* .....

Date of Order: / / *Telephone:* ( ) .....

number representing 1 of 3 types of address designations. This can be a specific address for a particular node; a general address for a group of nodes; or a broadcast address for all nodes on the network.

The second 6-byte address field contains the address of the Origination station. This is followed by a 2-byte Type field directive which identifies the user protocol associated with the frame.

The actual message is contained in the Data field, which has no fixed length. However, it does have limitations — it must be longer than 45 bytes and shorter than 1500 bytes and is automatically adjusted to accommodate the information being transmitted.

Ethernet also specifies an error-checking algorithm that calculates a sum of all the fields (except the Preamble). This result is stored in the frame check sequence field at the end of the frame. The receiving node makes the same calculations and compares its result with the frame check sequence number. If the 2 match, the system accepts that the data is valid.

The use of frames having a limited length prevents 1 node from hogging the whole system; the other nodes have an equal chance to transmit in the gap between. On the other hand, Ethernet's ability to use quite long frame-lengths, means that the system doesn't lose too much time in systemoverheads (addressing, checking and so on). There is also less chance of data collision as the system comes under load.

At the 'plumbing' level, Ethernet generally uses only 2 types of coaxial cable (thin and thick, or cheap and expensive — whichever way you want to look at it!).

The length of the cable is precisely stated as being a maximum of 2.8 kilometers in the Xerox standard and having no more than 1024 devices connected. But nowadays it is quite easy to link LANs to LANs and to extend the Ethernet system in distance, and in user numbers, through the use of bridges to other LANs through optical fibre, microwave transmission systems and infrared beams.

Coaxial cable isn't the only medium but it is the one defined by the Xerox and IEEE standards, and the one most commonly used. Ethernet's cable is marked every few feet at regular points; only at these points can nodes be added and removed from the system without breaking the major connecting cables.

In establishing a new node on-line, the output from the 'tap' at the mark on the cable, is fed to an Ethernet transceiver and on to an Ethernet controller — both of which are, again, tightly defined by the standard. Each node has its own controller which contains all the instructions necessary for correct framing and network control. They've now got all this equipment down to a three-chip VLSI set, which has bought the cost of the interface down to around the $100 mark.

Xerox has always seen Ethernet as an office automation network, primarily used to link powerful workstations based on Mac-like Smalltalk and Interlisp-D environments to DEC minicomputers. DEC has recently taken the lead in pushing Ethernet as a viable alternative to the new Manufacturing Automation Protocol (MAP) systems that General Motors has been attempting to introduce and standardise in the USA for large-scale manufacturing enterprises.

They've now got all this equipment down to a three-chip VLSI set, which has bought the cost of the interface down to around the $100 mark.

The problem with Ethernet in a manufacturing environment is that it can be subject to electronic interference when it passes large electrical and electronic machines. It is often too slow for some assembly line processes and the contention-type control mechanism means that timing delays in the system are uncontrollable and unpredictable. They say that Ethernet might not be able to react to a problem on the assembly line in the fraction of a second available before damage occurs — that's the opinion of the promoters of MAP, anyway.

On the other side of the coin, Ethernet is already well-established as a good, reliable LAN standard that is widely accepted around the world. It is predictable in terms of its reliability, offers no surprises and is cheap and easy to implement. Xerox and DEC point out that it will do 99 per cent of anything required of it at the present time and it gets better every day.

In the final analysis, this really is a computerised version of the story about the Old Bull and the Young Bull. My betting is Ethernet, the Old Bull — it takes its time but does the job thoroughly. □

Coming to grips with Networking - Part 3

Xerox may have succeeded in making Ethernet the de facto standard, but IBM are behind the Token Ring system which allows assignment of priorities to nodes and the ability to transmit digitised speech – but a lot of intelligence is needed to make a Token Ring system work.

ONE OF THE earliest forms of local area networking which was around almost before the term LAN had been invented, was the Cambridge Ring, so named after the British university in which it was devised. As the second part of the name suggests, the Cambridge Ring has a circular topology where information is presented to the group in a repetitive sequence — it's an informational merry-go-round.

The problem with ring topologies is that they are fragile. Any break in the circle, no matter how small and insignificant, and the whole LAN fails. It's a case of 'one out — all out'.

Star topologies provide a much better solution to this failure problem. LANs inherited the star radiating linkage system from PABX telephones. They are designed with one central controlling unit and independent wire pairs radiating out to all 'nodes'. With the telephone-based origins of the star network, it is not surprising to find that the main player in the field is AT&T with its StarLan system. (A general discussion of ring and star topologies was given in Part 1, October '87.)

A true star network needs a central node processor, but this is okay since any LAN above the simple level of peripheral sharing usually needs at least one dedicated 'file server' to control the common database, and in star networks this machine can double as the network controller. If one of the nodes breaks down, the system continues to operate but, of course, if the central control unit fails, the whole system goes down.

IBM established a dominant position in the office connectivity field fairly late in the development of LANs, although it did have an early release of the PC Network for groups with low data exchange requirements.

A Star-Wired Ring!

IBM announced in early 1984 that its major LANs system was going to use a hybrid of the star and the ring, a 'star-wired ring' topology, and then sat back for a couple of years while they developed the token ring technology to run on it. This is now the IBM Token Ring LAN which was only released in late 1985 and, as predicted, it has proven to be highly popular.

Star-wired rings have most of the advantages of both the ring and the star topologies. They don't need a central network controller, but neither are they subject to the 'one out — all out' problems of simple ring wiring.

Even though the cabelling radiates out from a series of central 'concentrators', IBM's Token Ring is 'logically' a circle. The network passes information around in a merry-go-round, with each computer in the system reading the frames of information and either acting upon that information or passing the frames on to its neighbour.

The key to obtaining the benefits of both the ring and the star elements lies in the use of a Multistation Access Unit (often called a 'Wiring Concentrator'), into which the radiating cables from the terminals/nodes connect. The specifications allow for normal twisted-pair telephone wires to be used, but recommends the use of special shielded twisted-pair cable. Fibre optics can also be used.

In the IBM system up to eight terminals can link into any one Access Unit, but Access Units are themselves linked together into the major network ring; this can have up to 260 PCs in the one system — plus bridges to other networks.

The key to overcoming the problems of the ring structure is the Access Unit, with relays which can automatically bypass a cabled connection by reacting to the presence or absence of a special test signal. The network is, therefore, able to instruct the Access Unit to isolate a device and drop it out of the chain if a fault occurs (or if the device is disconnected, or powered down).

If a break occurs in the ring, the next node downstream will react after not receiving a token for some time by sending out a special MAC-Control frame (discussed later) containing a 'beacon' signal with the address of its upstream neighbour. After a number of beacon signals have been received by the node, it will disconnect from the ring and the ring will automatically reconfigure.

The other major hardware component of the IBM Token Ring is the PC Adapter Card which plugs into the computer and

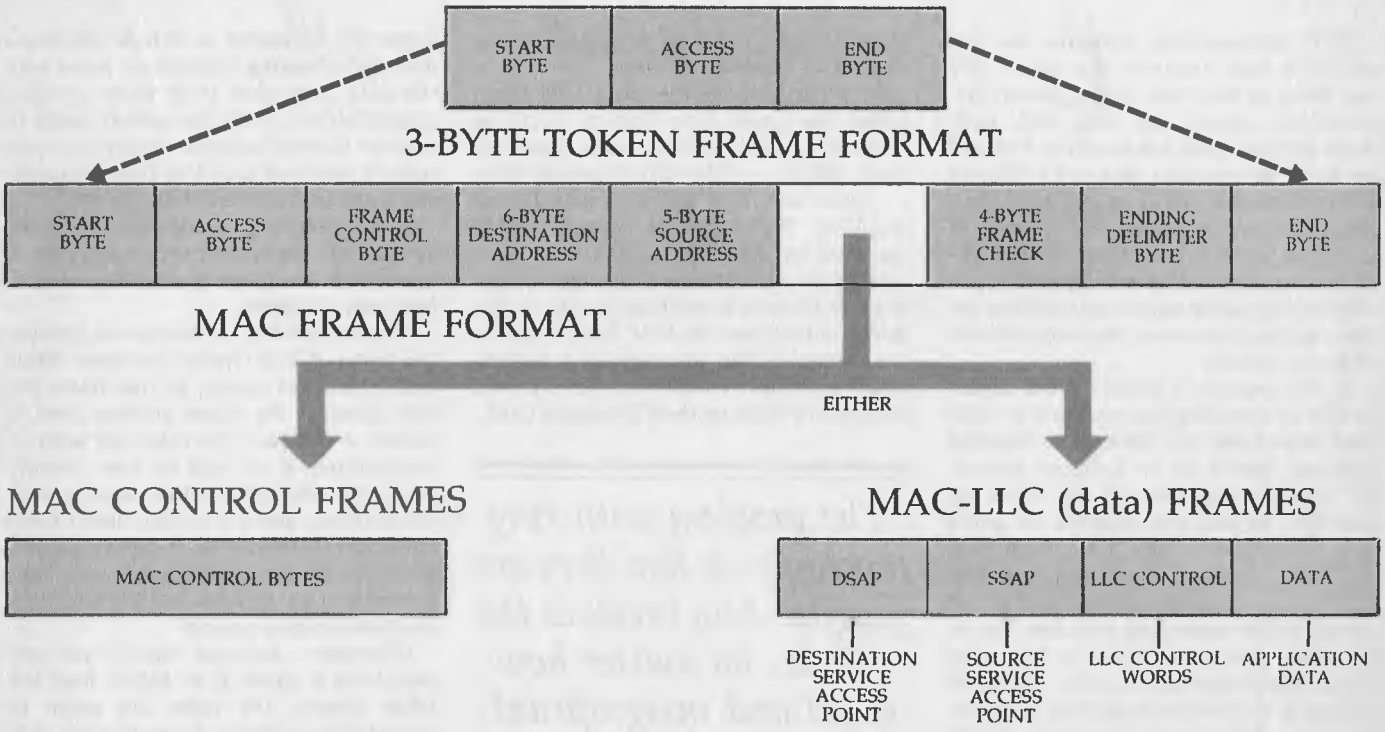


Figure 1. *The Token Ring Frame system.*

provides the logical link control functions and diagnostics for the system. The card contains a microprocessor operating under its own microcode.

It can transmit and receive data at a speed of 4 million bits per second, which sets the limits of the system since each node in a token ring system needs to read, reconstruct, and retransmit every frame of data as it travels around the system. Every node plays an active role with every piece of data on the system — unlike Carrier Sense (CSMA — see Part 2 for more information) systems where the nodes are passive until they recognise data addressed to them.

The real fundamental difference between IBM's Token Ring and Ethernet are the controlling protocols. Ethernet uses CSMA and IBM uses token-passing. IBM's game is very much like 'Pass the Parcel'; a 'token' (in this case a 3-byte frame of code) gets passed around the ring, and only the node holding the parcel can transmit. It's a sort of relay race, where only the man with the baton is allowed to run.

Ethernet's 'wait for a gap in the traffic' method is 'probabilistic', while token-passing is 'deterministic' in nature. There's no way of predicting when a particular node on an Ethernet line is going to be allowed to transmit — it takes its chances along with everyone else.

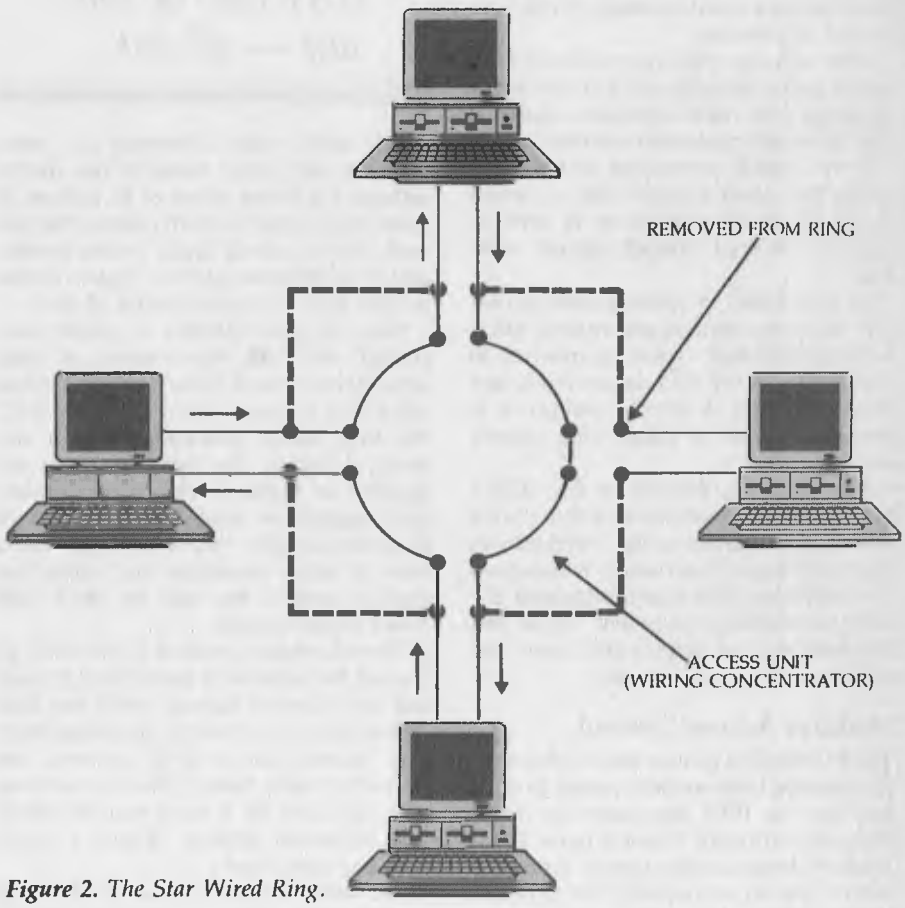


Figure 2. *The Star Wired Ring.*

With token-passing, however, we can enforce a strict order on the access. We can insist on the token being passed systematically around the ring, with each node getting equal transmission times. If we want, we can also choose to allocate priorities so that certain nodes have double, triple or quadruple the chance of using the system than others. With IBM's token ring network there is a system of priorities which allow important nodes to enforce access claims over the requirements of lesser mortals.

In the long-run, it could be this aspect of IBM's Token Ring that makes it win out over other LANs. It is difficult to transmit digitised speech on an Ethernet system, for instance, because you can never be sure that the next few syllables are going to arrive in time to keep the speech flowing. With token systems you can give speech the necessary priority so that the speech buffer never runs out. And the integration of voice and data is becoming increasingly important in LANs. The same objection to Ethernet is applied to Manufacturing Automated Protocols (MAP) which General Motors and other major industrial manufacturers are attempting to introduce as a world standard in machine control for factories.

Ethernet is the main competitor to MAP (which uses a token system but with a bus topology). The main objection, again, is that Ethernet's contention scheme means that the overall controlling system of a factory can never be sure that a control signal will reach a machine in time. It could be delayed through system overload.

So IBM's shift to token-passing protocols has some obvious advantages, but it is not without cost — mainly apparent in the expense of the PC Adapter Cards and the Access Unit. A lot of intelligence is needed to make a Token Ring system work.

IBM design is defined by the IEEE's 802.5 and 802.2 standards and it conforms to a couple of layers of the international OSI seven-layer model which is designed to standardise data communications between networking equipment. These two layers are also the same in IBM's own SNA communications layer model.

Medium Access Control

I don't intend to go into details about the competing layer-models, except to point out that the IEEE Standards for Token Ring only define the Physical Layer (Layer 1) which designates the type of wiring system to be used, and a part of the Data Link Layer (Layer 2) called Medium Access Control — or MAC, for short.

A second part of the Data Link layer, called the Logic Link Control (LLC) is common to all three IEEE LANs specifications (802.3 — CSMA/CD Ethernet, 802.4 — Token-bus MAP, 802.5 — IBM Token-ring IBM). The Logic Link Control itself is specified by IEEE 802.2. This MAC level subsection of the Data Link layer, specifies how the access method used by a LAN works. In this case the MAC level specifies the control of the token-passing system, which exists as firmware stored in the 16 kilobytes of ROM on the PC Adapter Card.

The problem with ring topologies is that they are fragile. Any break in the circle, no matter how small and insignificant, and the whole LAN fails. It's a case of 'one out — all out'.

This MAC control firmware can communicate with other nodes in the system without you being aware of its actions. It does this in order to both control the network and to correct faults in the system, and it is therefore given a higher access priority than the transmission of data.

The LLC level software is mainly concerned with the transmission of data around the network. There are actually two types of LLC frames specified by the IEEE. The first simply packages the data and sends it out on the network (errors are handled by higher levels), while the second expects an acknowledgement back from the receiver. This second type has a form of error detection and correction built-in, and is the way in which IBM Token Rings operate.

The information needed by the MAC to control the network is distributed by special MAC-Control frames, while the data distribution is handled by, so called, MAC-LLC frames (since MAC controls are needed on every frame). This sounds complex, but don't let it worry you, its not all that important anyway. (Figure 1 might help you sort it out.)

We saw in November how the Ethernet frame was formatted to include addresses and error checking information alone with the data. But token rings have a further complication in that the system needs to transfer control between nodes in a systematic way, and to do this they use a special three byte frame called a token.

This token frame circulates around the system in a predefined order, and part of one byte in the frame is a priority number from zero to seven.

When a terminal on the system receives the token, it first checks the token frame priority number against its own frame priority level. If the token priority level is higher, it will pass the token on without transmitting. It can add its own 'priority request' to the token before passing it on if it chooses, and if a change hasn't been made to this request as it passes around the ring, the originating node will take control and transmit its data the next time the token comes around.

Whenever a terminal frame's own priority level is equal to, or higher than the token priority, the node can begin to transmit immediately. A terminal will only release the token frame after it has finished transmitting, or after a predetermined time or set number of frames have been sent.

What actually happens is that the first and last bytes of the token are retained, and the terminal's information bytes with addresses, data and error checking, are inserted between these token 'book-ends' and sent on their way around the ring.

The receive part of the system listens all the time, and it can detect the returning header of its own information, while still transmitting the 'tail'. It checks this returning frame to see that it has been copied properly by the destination node.

At this stage, a new three-byte token is regenerated and passed off to the next terminal in line. This process ensures that there can only ever be one frame on the network at any one time, and it applies both to MAC-Control frames and MAC-LLC frames.

When a new terminal is connected to the ring it immediately sends out a special MAC-Control frame which includes the address number it proposes to use as identity. If this address isn't already in use, the frame is returned intact and the initialisation process begins.

The proposed number then becomes the terminal's network address and is used when sending or receiving frames. Incorporated into this address is a function code which identifies those nodes on the system that play special roles.

Ring Poll

One special form of the MAC-Control process worth mentioning is the Ring Poll which is initiated by the MAC-Control frame and causes each node on the system to transmit to its downstream neighbour. This node checks the incoming address against the one it has stored, and so detects whether unnoticed system changes have resulted in a new upstream neighbour. If it finds a discrepancy, it advises the Network Manager, if one exists in the system.

Small token ring systems don't need a special network control computer, but as they grow larger this function can be taken up by one member of the circle. Later still, when the system demands it, a dedicated network managing computer is usually installed.

The Network Manger is responsible for keeping a list of all nodes currently using the network, and it controls the insertion and deletion of nodes. This computer becomes the central recorder of ring status. Any node on the system can communicate with the Network Manager by using a MAC control frame with the Manager's node address incorporating the special function code. Without this code, the Network Manager terminal is just treated like any other node on the system.

The problem with ring topologies is that they are fragile.

Two sub-sections of the Network Manager control special functions. The Ring Parameter Server controls the insertion of a new node into the system and provides it with its addresses and priority codes. The Ring Error Monitor collects information about errors on the system and sends reports to the source address for correction. Error detection and correction is a divided responsibility in the larger token ring system. If there is a Network Manager on the system, then it will bear the major responsibility — if not, the role is played by the Active Monitor which can be any one of the terminals on the system.

After a network failure, or when the LAN is first fired up, all nodes actively contend for the role of Active Monitor by sending MAC-Control frames that claim the role. Usually the node with the highest address wins. The Active Monitor looks after the circulation of the token, and it sends reports on token errors and changes to the ring status if a node is inserted or deleted to the Network Manager. It monitors the system every 10 milliseconds, mainly to check the token for problems. If it finds a problem with the token, it will immediately purge the ring and send out a MAC-Control frame to check whether the ring is still functioning. This is the only case where a frame can be sent by a node without possession of the token.

If this MAC-Control frame travels around the ring and returns to the Active Monitor without error, a new token is created and set in circulation. If an error is returned, the Active Monitor will keep transmitting control frames until the faulty unit is isolated and removed from the ring by its Access Unit.

As you can imagine, the complexity of operation of the Token-ring system means that quite complex software is also needed, but I don't propose to go into this in this series. □

Coming to grips with Networking - Part 4

Broadband or baseband has been the choice in the past for a networking system – but now there's also Metropolitan Area Networks, Central Office LANs, fibre optic and infrared networks to come to grips with. Let's connect the possibilities . . .

ONLY A COUPLE of years ago the main argument in the networking world was between the competing camps of the broadband and baseband advocates. That's hard to imagine since there's little argument now: baseband has had a resounding victory. However the past popularity of broadband has left perhaps 25 per cent of total LANs around the world as broadband systems, and what's more, broadband could be making a comeback.

Broadband didn't die out: it found a niche in providing LANs 'spines' or 'backbone' links where it was not in direct competition to the baseband Ethernets, AppleTalks or IBM Token Rings.

It also became the LAN of choice for large scale networks where enormous amounts of data need to be transferred constantly. In the future it will possibly also establish itself in large scale manufacturing as the way to link assembly line robotics with the central controlling computers. This growth may parallel the boom in computer-aided design and parallel processing applications which typically require channel-widths of between 300 and 400 Mbps.

Broadband LANs use the same cable as the American cable television industry (the so-called CATV — Community Antenna Television) wiring standard. This is a 75 ohm coaxial cable that can be divided into multiple channels for simultaneous video, data and voice transmissions — and here is the secret of a possible long term broadband revival.

Connecting people

At present we view LANs as a way to connect computers, but this idea is evolving into the concept that LANs are a way to connect people. During this evolutionary phase we will gradually be introducing vision and sound communications into our networks along with the data. Baseband networks generally will not be able to make this change — most just don't have the capacity to go beyond systems that are mainly data with possibly some minor voice component.

Broadband signals are propagated by analogue techniques which are similar to the way modems and audio frequencies are used to send data down phone lines. These broadband coaxial links, however, transmit data at multi-megabit rates and use radio frequencies, not audio.

Existing analogue equipment and techniques can be applied to broadband LANs systems, which was the main reason why broadband was very popular early in the development of networking. Many broadband LANs use slightly modified off-the-shelf equipment such as video amplifiers, video cable taps, splitters and a special piece of hardware called a signal translator which takes the analogue signal at one frequency and converts it to another on the same channel.

Most broadband systems use the 'mid-split' technique to allocate the radio frequency spectrum on the coax. With mid-split, approximately half the available bandwidth is reserved for data flowing 'downstream' and the other half for 'upstream' communications, with signal translators making the conversions at each end.

With mid-split you only need one cable, but Wang and a couple of smaller broadband LAN suppliers use a two-cable network where one cable is reserved for each data direction.

Either way, the cabling needs of broadband are costly and there is the additional expense of digital-to-analogue conversion equipment and multiplexers. Despite these costs, the sheer amount of data that can be carried by a broadband LANs usually justifies the expenditure.

Other factors favour broadband also. Analogue LAN equipment is tried and tested — it has been used for many years under adverse conditions in cable TV systems around the world and it has a very low failure rate.

Consultants also have personal reasons for advising companies to take the broadband route — they like to keep eating. Experience has shown that it is almost impossible to predict LANs usage in an office or factory environment for more than a year after introduction, so it makes sense to suggest a system which can take

two or three times the load estimated, rather than one that is close to limits. To paraphrase an old saying: 'No one ever got fired for buying a LAN with excess capacity.'

With any LAN, the bandwidth available decides the number of channels and the data rates (and therefore volume) that can be transmitted. Depending on the methods employed, broadband channels can have hundreds of kiloHertz (or even megaHertz) of bandwidth, so the overall systems can have data capacities beyond the wildest dreams of baseband LAN designers. Even a low capacity broadband network will carry perhaps ten times the traffic of Ethernet.

New baseband LANs

Mind you, new baseband LANs are being developed to carry data at much higher rates, and the problems that bog down contention systems like Ethernet are being overcome by new network access techniques.

IBM scientists in Zurich are already showing a 16 Mbps Token Ring network, and exploratory work is in progress on still higher rates — in the order of 100 Mbps. At these speeds, voice and data signals can be mixed on the same line so that users can send, view and discuss information simultaneously through the same terminal.

But at best, it will be very restricted mixed-media capability — it's hard to envisage multiple video, voice and data channels for simultaneous video conferences or even voice discussions on any single baseband network.

Digitised speech data must be transmitted in a reasonably continuous stream with a guaranteed rate of about 32 Kbps for telephone-quality voice, if it is to preserve the inflections and tonalities of human conversation. IBM's Zurich experiment with Token Ring involves special circuitry in every node that provides priority to the speech component. Whenever speech prefix bits are detected, the LAN cuts off all data transmission for a few milliseconds and gives right of way to the speech data.

Data is interleaved progressively with the priority speech bits, but only at a rate that ensures speakers and listeners aren't aware of any interruption. Obviously this speech priority restricts the flow of data through the network, but the assumption is that voice will only occasionally be used.

There are further mixed-media problems with baseband contention systems like AppleTalk and Ethernet: speech (or video) can't be transmitted through networks which have random delay patterns — beyond the limits of the video or audio delay buffers, and the frustration level of the people communicating.

This limitation has faced scientists working on super-fast Ethernet systems because the IEEE standard uses 'probabilistic' rather than 'deterministic' controls on its communications. A voice transmitting node never knows for sure exactly when it will get access to the network.

There is a French variation of the Ethernet standard which is heading in the same direction as IBM's Token Ring. The French Government's INRIA research laboratory has come up with an algorithm for making Ethernet more responsive to heavy loads by adding deterministic controls to the communications. They've dubbed this the IEEE 802.3D (for deterministic) standard.

It changes the way Ethernet operates, but only at the cost of replacing one chip on the standard Ethernet PC boards. What INRIA have done is to replace the algorithm which defines the random time-out delay that each node must make after a data collision.

INRIA's algorithm limits each node's attempts at retransmission, since these constant collisions when fighting for network control are the cause of Ethernet's rapid drop in data throughput under load. Now after a collision, only half the nodes on a network are allowed to re-transmit (thereby halving the chance of a second collision). If another collision does occur, the network halves the number of active nodes again, and then again, if necessary.

The nodes drop in and out of contention themselves by counting the distance they are away from the transmitting node. They know how many are nodes are on the network, and so a simple calculation lets them decide whether they can contend or not. So this complex control procedure is done without the need for an overall network controller — thus maintaining one of the strengths of the Ethernet system.

With this new approach, a worst-case message delivery time can be guaranteed within clearly defined limits, and this allows Ethernet to deliver voice, and also to be used for factory automation networks. But it doesn't increase capacity by much — it just introduces an automatic form of arbitration when Ethernet is reaching capacity.

Broadband LANs are obviously much better than baseband in carrying out both voice and video — and they can even piggyback baseband networks, if required. Ethernet for instance, is quite often carried for long distances on one channel of a broadband network.

Broadband didn't die out: it found a niche in providing LANs 'spines' or 'backbone' links where it was not in direct competition to the baseband Ethernets, AppleTalks or IBM Token Rings.

Another factor in broadband's favour (and the one that makes it especially useful for large-scale manufacturing industries), is that analogue technologies suffer less from electrical interference than digital. Therefore, they are more effective in industrial plants and other areas where electrical noise is a factor. And, since the amount of noise on a line is generally proportional to distance, this also means that broadband LANs can stretch over longer distances.

In the US, cable television systems are usually city-wide, radiating out for distances up to 50 kilometres from the hub, and it is not uncommon to find a broadband LAN extending its reach the same order of distance. In many parts of the US, the local CATV system has taken on a new non-television role in the past few years.

MANs and CO-LANs

Cable operators now offer channels on their systems for broadband data distribution. The Americans have invented the term MAN (Metropolitan Area Network) to refer to this dual use of the cable network. We don't seem to be developing MANs in Australia, mainly because we don't have cable television, except in the Adelaide foothills and the coastal Sydney region, neither of which are noted industrial sites.

The American telephone companies are also getting in on the act by offering CO-LAN (Central Office LANs) through the telephone network — but they don't have the coaxial links that the CATV companies already have installed. AT&T are selling a 56 Kbps Datakit Virtual Circuit Switch CO-LAN package to the 22 Bell operating companies, and some smaller telephone equipment suppliers are developing CO-LAN packages of their own.

CO-LANs are generally data and/or voice and data network services which are based on switching equipment located at the telephone company's central office rather than on the customer's site. Ameritech is working on a CO-LAN which supports user data rates of up to 2 Mbps on existing twisted-pair telephone cables. They claim that their CO-LAN can handle Ethernet LANs, basic RS232C asynchronous interfaces, IBM 3270 synchronous terminals and X.25 packet-switching. They expect to add ISDN to this list in the near future.

Fibre optics

There is also a growing interest in fibre optics for long-haul LAN applications, although only a very few percent of current networks in the world use this media — and then generally only as a backbone to connect baseband systems.

Optical fibre has the capacity to carry voice, video and data at gigaHertz rates, but most optical fibre backbones are used only to support a simple exchange of data between Ethernet systems chugging along at less than 1 Mbps — which is overkill by a factor of 50 or more.

Part of the problem in the acceptance of optical fibre as LANs media is a lack of international standards for optical fibre links. The IEEE's 802.8 technical advisory group is still wrestling with a standard specification which will include the core-diameter size, physical links and network topology.

Optical fibre offers low bit-error rates ($1:10^{12}$), bandwidths in the Gigahertz range (per fibre), zero electrical interference and complete electrical isolation. Security is also a consideration — it is hard to tap into optical fibres, and for this reason almost half the optical fibre LANs installations in the world are for defence uses.

The isolation from electrical currents and noise could well be the major factor in promoting optical fibre systems into industry — especially when coupled with MAP (Manufacturing Automation Protocols). Many people think that the use of glass media is essential in large manufacturing plants to take care of the electromagnetic interference problems that plague copper-wire systems.

The negative side of optical fibre is in the limited number of direct connections you can make to a length of fibre. Generally, 24 or so taps is as much as a simple system can handle because of dirt and scratches that create problems around each splice. It is easy to make good splices in a laboratory, but quite another thing to make them in a dirty office or down a manhole in the street.

Infrared

Another relatively new development in LANs transmissions is in the use of infrared, which solves the problem of wiring systems by discarding wiring altogether. There are point-to-point infrared links which are used to replace cables between semi-adjacent buildings, and also broadcast systems that have application in work-group LANs.

These techniques allow the whole of an office to be tied together by light beams and they avoid the cost of cabling — an important factor in some buildings, especially where changes are constantly made to the position of nodes.

The idea is to bounce a high-intensity modulated infrared beam off the ceiling of an office and allow it to scatter around the room. It can be received by any network node using a simple solid-state detector. This node, in turn, can bounce its own message against the ceiling through a solid-state infrared emitter and broadcast it to all others in the office.

Infrared seems to reflect around an average office space quite well, and you can cover quite an area with surpsingly little power. Collision detection systems seem to work best with infrared broadcast techniques since there is no physical sequence to the links for token-passing — although a limited form of token-passing is possible. □

Coming to grips with Networking - Part 5

Large networks are soon to cover whole continents and these will be linked by packet-switching services – mainframes and PCs will then be peripherals to the network!

IN TRYING TO EXTRACT any sense of direction from the booming local area network (LANs) market at present, you would have to say, sphinx-like, that LANs are both moving towards wider connectivity and towards workgroup solutions. (There's nothing like having it both ways to increase the accuracy of your prediction!)

Apple originally promoted the workgroup concept many years ago with the idea of Macs linked by AppleTalk to other Macs in the immediate computing neighborhood. In almost all offices the bulk of data exchange is logically and geographically confined to a small group, so AppleTalk's 32-node limit was not a problem.

The problem with AppleTalk was that it was released too early – it was on the market far ahead of good multiuser applications software. Generally, there was also thought to be a conceptual conflict with AppleTalk, in that the 'workgroup solution' didn't fit the prevailing 'bigger is better' philosophies of LANs design. In time, both ideas have proved to be compatible.

Now with its AppleShare server and connectivity to DEC minis and other larger and faster LANs systems (both Ethernet and IBM's Token Ring), AppleTalk is beginning to show its extension capabilities (and they are quite impressive) while maintaining its workgroup basics.

AppleTalk is not alone in making these changes. Ethernet now runs on optical fibre and broadband systems and IBM's Token Ring has come out in bus and star topology versions and now has the ability to run on fibre optics, piggybacked onto broadband analog frequencies.

In fact, network designs and the networks themselves have proved to be dynamic, not static. Once installed, there are always new connections to be made, new facilities to bring online, equipment upgrades, mainframes to be connected, and changing relationships within groups in an organisation that require the interlinking of nets, often of different LAN types.

Most office networks start out with six or seven PCs sharing a printer; later they add both shared files (through a file-server) and additional peripherals. Before long, the group is experimenting with Ethernet backbones to link other independently evolved networks, and they want gateways to the company mainframes and communications links to the outside world. What was a simple peripheral-sharing LAN has become part of a mega-LAN or internet.

This is what happened to Digital Equipment Corporation, the main promoter of Ethernet. It now has 15,000 nodes on its own Internal network and they get requests for over a thousand new connection each month. The French Post Office has the same problem – it is now finalising a 30,000 node LAN based on Fox's 10-Net. It is hard to apply the local area concepts to networks of these sizes – but they grew from basically workgroup nets.

Networking, for many larger organisations, is no longer an option but part and parcel of its data processing system. PCs are information processing machines, but they only realise their potential when they have large stores of information available (from hard disks, online databases, and now DC-ROM) and can exchange and co-operatively update this information (through PABXs and/or LANS).

Changing concepts

Before very long we are going to see the geography of the larger of these networks covering whole countries or even continents, with most being permanently linked to packet-switching services which exchange data across the oceans. Both the data and the computing power will be distributed and belong to the network as a whole; the mainframes and PCs will be peripherals to the real computing device – the network.

These changing concepts are having a profound effect on the way LANs technology is evolving. A few 'standard' network types have been accepted, but constant developments wreak havoc on the standards. Different vendor implementations of essentially the same technology may be quite different; not all 'like' LANs are alike. However the LAN market is rapidly becoming the inter-LAN market, which is forcing the vendors back in line.

In time, the present limitations of distance, topology and data throughput will eventually be overcome – although new problems of security and reliability arise. The original Ethernet specification called for a coaxial cable backbone which was divided into three cable segments of 500

meters each. There are two cable types – 'thick' for long runs (backbones) and 'thin' for shorter distances (office connections).

The cable segments are joined by active repeaters, so the maximum length of an Ethernet Version 1 network is 1.5 kilometers (now 2.8 kms). This includes the backbone length and the length of the cables that join the stations to the backbone, calculated on a formula that takes into account the thickness of the wire used.

These distance limitations are due to a propagation delay of 9.6 microseconds, with data moving from one end of the cable to the other. Even at the speed of light it takes time for data to move along a length of cable, and collisions can only be detected when the signals collide, rebound, and the jumble is again read at the remote nodes. Remember that the system is operating at 10 million bits per second, so roughly 200 bits can be travelling on the longest Ethernet before evidence of a collision is returned to a node.

Repeaters can double the propagation delay, so the 'collision window' is defined as being twice the maximum network 'lag', and this effectively establishes the theoretical maximum carrying capacity of a system.

As networks grow in size the problems are compounded. The Ethernet (802.3) spec restricts the number of stations on a network to 1024, but some network designs circumvent this by employing buffered repeaters to regenerate the transmitted signals.

We need to clarify the LANs terminology here. There are four different network components with roughly similar (and sometimes overlapping) functions: repeaters, bridges, routers and gateways.

Repeaters

LAN repeaters are very similar to those use by OTC to regenerate voice and data signals on submarine cables under the oceans – and they are generally cheap and reliable.

Repeaters deal only with the physical layer (Layer 1) of the OSI model, and they simply 'repeat' every packet on a LAN. They read all packets on one network segment, amplify them, and transmit them on another. They don't discriminate in any way because they are dealing with the data in a purely 'bit' form without any attempt at network management.

However, buffered repeaters are able to overcome the propagation delay problem because they hold the incoming packets in a buffer and only retransmit them into the other segment when an opportunity presents itself.

On either side of a repeater the segments constitute one single LAN – both physically and logically. All traffic is present on each segment – with one exception – some repeaters now contain logic to prevent problems in one segment form effecting the others. These repeaters won't retransmit erroneous signals.

Repeaters are not only used to link segments in a straight line, they can also be used to create internet topologies with a tree-branching structure (but without loops). Digital make an Ethernet Thinwire (backbone) system with a nine-port repeater which allows you network to connect to either other Thinwire cable segments of up to 185 meters. You can add 29 devices to each segment for a maximum of 232 stations per multiport repeater.

You can also get optical cable repeaters that allow the maximum distances between Ethernet segments to be extended to over a kilometer – but that's only part of the story since you can then begin to link different networks – and here, the sky's the limit.

Bridges

Bridges allow us to link two LANS of the same type so that they are physically separate, but logically the same. One device on the resulting internet can address another as if it were on the same LAN.

The distinction between repeaters and bridges lies in the fact that bridges filter and discard all packets that are destined for the local side and regenerate only those packets addressed to the remote side.

All network packets contain both a source and a destination address. The MAC (Media Access Control) rules define the way these packets can access the physical media: this is part of OSI Layer 2 – the Data Link layer and it is at this level that internet bridges operate.

Devices on each side of the bridge simply address packets to each other as if they were on the same LAN – never knowing that they are dealing with a bridge since the system is 'protocol independent'. Bridges function without the need to use protocols beyond the standard Data-Link layer addressing; they route packets strictly on the basis of their destination address without using any routing protocol.

However, a smart bridge 'learns' which device is on which side, by monitoring the source addresses. It uses this information to build and update its 'routing table' and checks this to decide what packets to filter

FILE SERVERS

NETWORKS NEED to have access control on any database (or other files) used by more than one person. The file-server must differentiate between private and public 'volumes' (virtual spaces).

Public volumes might also need to be made 'read only' to most users, but someone will need to update them, so you must have a system of access rights. Usually these rights are assigned to passwords (attached to the user), but different PCs can also have unique reference numbers held in ROM onboard the net-card (attached to the device)

System software must also have a way of preventing others from accessing a file that is already in use. There are no problems with a standalone PC because you make changes to a file by transferring it (from disk) to your own machine memory, updating it, then copying it back. Sometimes the file can be away from the disk for a long period of time – but this doesn't matter to a lone user. However, in a network environment anything could happen to the disk-based version in that time if the file isn't locked.

The file-server's job is to keep track of where files are in the system, and differentiate between those users who have access on a read only basis and those who can make changes. It must also distinguish between 'acquire' and 'access' requests.

When LANs are used to link a number of terminals to one large database so that a master file can constantly be updated by all operators, it makes little sense to lock the whole file while each record is being updated. Everyone else on the system will be forced to just sit around and wait until the file become available again.

The solution is to only lock the individual records, but this is more risky than file-locking. The type of processing involved in record-locking is largely dependent upon the operating system of the peripheral PCs, and it is for this reason that the changes made recently to DOS are as important.

and which to forward.

Bridges deal with data at the packet – rather than at the bit-level, and they also apply a network management role.

Two separate LANs linked by a bridge will take care of overload conditions much better than two segments joined by a repeater. Most traffic will remain within a local area, so the smaller the LANs unit, the less effect a flood of transmission from one node will have on the availability of the internet to others. Breaking a big LAN into two smaller ones with a bridge between improves reliability and security.

Router

OSI Layer 3 defines the network as a whole – and so it is at this level that the rules used to interconnect complex networks are spelled out – in particular items such as the internet addressing systems, like the widely used US Department of Defence Internet Protocol (IP).

Any device wanting to distribute internet packets through a router must use the same internet protocol and explicitly tell the router to forward the packets by addressing it as a device. Data is transferred 'intelligently' on a message basis. A router simply passes on any incoming packets but it does not filter or monitor the general LANs traffic since it receives internet traffic only.

Routers are much more intelligent than either bridges or repeaters, and LANs which are joined together through a router are physically and logically separate. Routers become important when the network consists of complex LANs with multiple communications paths, perhaps with loops and substantial geographical distance.

Gateways

LANS gateways have taken off this year because of an almost religious 'conversion' to networks by mainframe DP personnel. Gateways exist to allow communications with facilities outside the network or the internet.

The most common gateway is one used to provide X.25 communications over public packet-switched networks like Austpac and Data Access.

Gateways operate at even a higher level than routers. They generally provide not only the physical link but also some kind of applications connection. Some require the use of a dedicated machine (which avoids potential bottleneck problems) while others use a garden-variety PC with special software.

Gateways also provide protocol-conversion between the LAN and a network of a different type, or links with long-haul architectures such as SNA or X.25. Designers are now focusing on the connection of different micro-based LANs and on the provision of gateways to the larger computers through 'virtual' networking. With this approach, a PC-user can access resources on both local and remote servers as if they were within his or her workgroup.

Although virtual networking is maturing, it has its problems. At present, workstations on a virtual network tend to access minis and mainframes as terminals. What we have today is essentially only a distributed filing system using database managers rather than true distributed processing with a wide range of applications.

There is still an enormous amount of work to be done on LANs software – especially where gateways are involved. IBM's newly announced LU6.2 peer-to-peer protocol may be one solution, but it is not

yet available. Apparently it can be implemented on a server with a micro-interface on one side and mainframe-support applications on the other.

LU6.2 was designed for long-haul leased-line connections rather than for LANs and it is intended primarily as a link between user-nodes on a SA network, (which is mainly used in America).

The aim of the International Standards Organisation's OSI layered architecture is to make these systems protocol-independent. LANs are hampered by the incompatible protocols used by the different vendors, but the ISO is still working on its layered model and only a few of the lower layers are fully defined.

The components necessary to make distributed processing work on large internets – such as distributed file systems and network 'operating systems' or network services – are just beginning to emerge. Sun Microsystems' Network File System (NFS) and AT&T's Remote File Sharing (RFS for Unix System 5) system work in the upper levels of the OSI model – and these are still in confusion.

The network operating system's primary tasks are to know where everything is on the system, and to control the resources. These new large network tools will need to be complex and layered because each computer on the network will have its own operating system which will need to slot in to the network O/S at different levels.

NFS is both machine- and operating system-independent, and it allows transparent access to files on other LAN systems. If, for instance, you are using PC-NFS with an IBM or compatible and Ethernet, you can enter DOS commands and transparently access files on any other linked computer running NFS.

Internets

The move towards internets requires much more complex networks operating systems than we have at present. Many are in the pipeline. In the meantime the mega-LAN vacuum is filled by a number of high-level protocol suites including Xerox's XNS, the US Department of Defence's TCP/IP, and Decnet from Digital.

The two most commonly used mega-LAN protocols are the TCP/IP and the Xerox Network System (XNS). These both establish ground rules that allow different vendor products to communicate with each other. Also important are IBM's Network Basic I/O System (Netbios) and Advanced Program-to-Program Communication (APPC) which present a common set of communications standards for multi-vendor products on the Token-Ring LAN. There are only a few LANs which support Netbios fully.

Netbios is said to be easily incorporated into third-party token ring devices, where it ensures they can operate together. The IBM Token Ring has a Netbios interface (which was first introduced for its PC Network), plus a SNA version of APPC.

In the OSI model, APPC extends to Layer 7 while Netbios sits conceptually between Layers 5 and 6 – defining the interface to LAN communications subsystems and detailing call sequences and control tables.

TCP/IP (Transmission Control Protocol/ Internet Protocol) is a much older dinosaur that is still gaining new advocates. Novell and Micom-Interlan have both recently announced new gateways and software products using TCP/IP; at least partly because the protocol is essential for sales to US government and engineering marketplaces. It is byte-oriented and is slowed by the need for mandatory checksums – but it has a virtual-terminal protocol.

Some universities in the US are using TCP/IP to link hardware ranging from Cray supercomputers down to desktop micros. Ethernets also tend to use TCP/IP to communicate across a range of computer systems on a network.

TCP/IP allows diverse hardware and software products to communicate with each other. It had been in the Public Domain for nearly a decade when it entered the commercial area on the back of the Unix 4.2 operating system. The University of California at Berkeley incorporated TCP/IP into its Unlx kernel along with an address resolution protocol (that map TCP/IP addresses to Ethernet IEEE 802.3 addresses and thereby makes a convenient interface).

XNS is more efficient that TCP/IP due to its packet-orientation. Xerox designed XNS for Ethernet LANs about seven years ago and it is now the defacto standard for interLANs in office automation although it is still not fully standardised. Novell has a Netware version called IPX for 'Internet Packet exchange.

Both TCP/IP and XNS are layered protocols which can be related to the OSI model. Both correspond to Layer 1 and IEEE 802.3, 802.4 and 802.5 standards. Both offer compatible services at OSI layers 2, 3 and 4 while TCP/IP goes on to layers 6 (and 7) with the Telenet File Transfer protocol that allows data transfer between dissimilar products or operating systems.

So the software components we need for efficient distributed processing and data exchange are gradually falling into place, and national and international mega-LANs are becoming a reality. But a critical question arises: We can build it, but can we manage it?

Large distributed networks have different management problems to small workgroup LANs – not the least of which is finding the device you wish to contact. Artificial intelligence and knowledge bases will probably have to come into the picture simply to handle the complexity of information needed for addressing and using remote host devices.

Addressing

If every device on a mega-LAN is to have a unique address, then the number of bits allocated for addressing determines the logical maximum number of devices possible. In early CSMA/CD and token systems the standard was set at 16-bits which allowed theoretical possibilities of up to 65,000 devices. But now the address size has been extended on most systems to 48-bits which provides 281 trillion unique addresses – enough for everyone, everywhere in the world to have a unique number and then some.

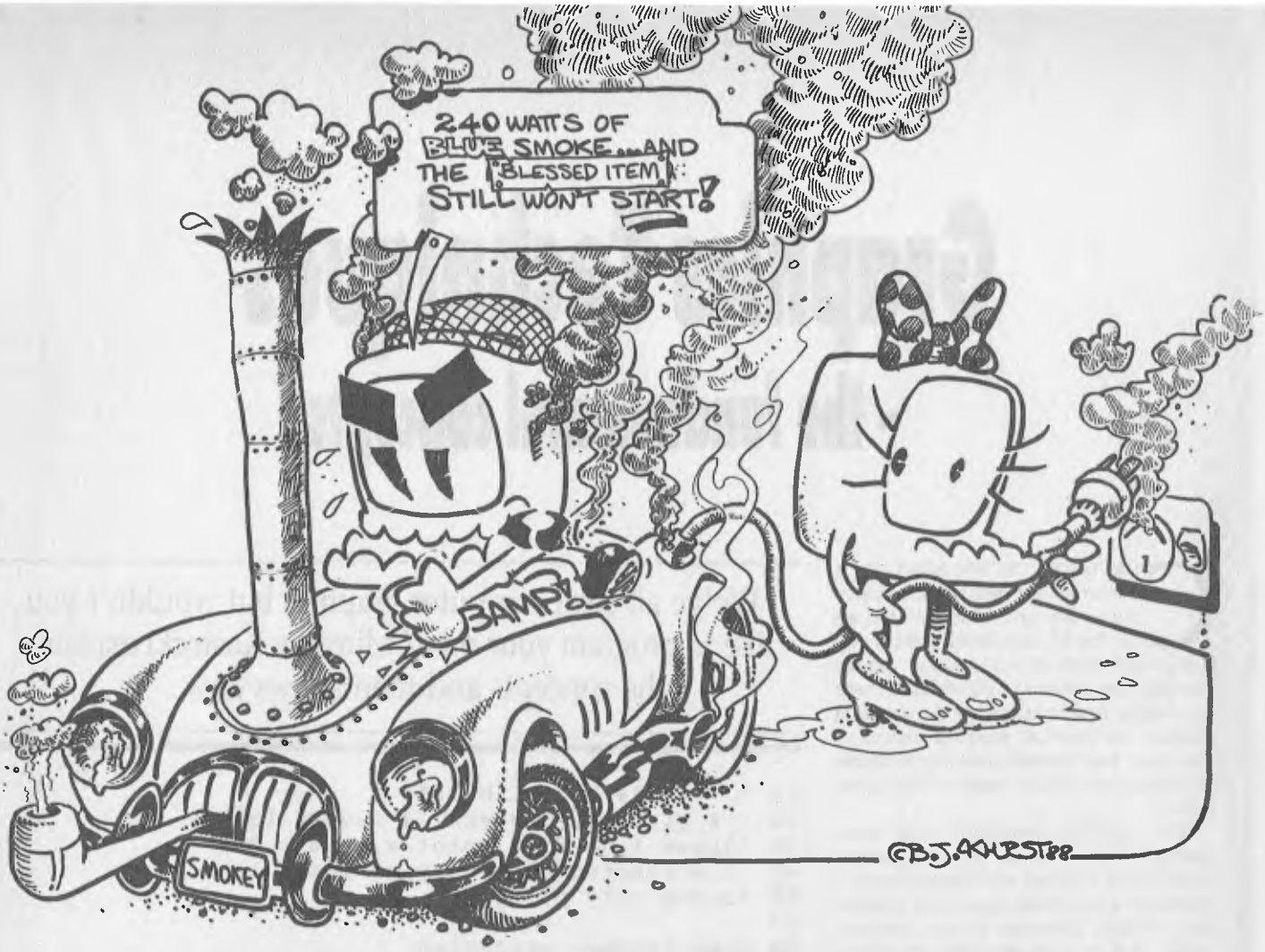
Addressing becomes vitally important as networks grow – not only in the number of devices that can be contacted, but also in the range of addressing options. And since it is in an evolutionary stage, Ethernet supports three station addressing schemes: network specific, unique and multicast.

Network-specific addresses are unique to stations on their own network only, but the same address may be used on another network. To address these devices from outside, you must specify both the network and the address.

Unique addresses have no twin on another network. This is important in mega-LANs communications because it is not necessary to specify both a station and its network to produce an unambiguous address at the next protocol level.

Multicast addressing is a facility where packets may be targeted at more than one destination – in other words, for distributed applications. Broadcasting is the extreme case of multicasting where packets are sent to all stations. On Ethernet the broadcast address is simply a series of 48 ones.

Another valuable addressing facility supported by some LAN vendors is resource, naming. Users assign specific names to resources and call them by invoking the name rather than the network address. When we start getting 218 trillion unique addresses, some such simple translation system becomes essential.□



NO SMOKING!

Ewart Stronach reveals the workings of a computer . . .

FINALLY, I have it figured out. All this technical rubbish about bits and bytes, millivolts and megawatts is simply a ploy, devised by the early technical workers to protect their opulent lifestyles. 'Bury the task in intricate terminology,' they said. 'It'll be years before anyone finds out and we'll all be rich.'

Their secret is out – it's *smoke*. Computers run on smoke! All those little wires and tracks simply carry smoke from one component to another. If one of the wires or tracks breaks, the smoke gets out and the computer stops working.

The smoke is made in a device which the technical people call a power supply and stored in devices called capacitors. When it is full up, it starts to leak and you have to go to an expert to get a new one.

The bigger the device, the thicker the wire needed to carry the smoke, or in the case of a complicated device, many thin wires each carrying a small amount of smoke. Devices which open out to the front of the computer are very prone to leaking smoke and any sign of such a leak from your disk door should be viewed with grave suspicion.

The gravity of the breakdown may be judged by the amount and colour of the escaping smoke. Heavy black smoke means heavy repair bills, while light grey smoke is usually the cheapest.

The heaviest black smoke comes from mains powered devices which are connected to the lights in your house. When you turn on a light switch, the smoke flows into the light bulb with such force that it gets excited and glows. This sucks up all the dark in the room and turns the smoke black. This black smoke goes direct to your toaster and either falls on your toast or leaks out the back.

Smoke travels in one direction only and transistors and ICs are simply devices for shunting the smoke about. If they fail and smoke goes along a wire against the normal flow, it almost always bursts out the side of the wire.

Armed with this most basic premise, you may now confidently carry out your own repairs and all you need to know is which direction the smoke should travel.

This has got to be one of the best kept secrets of modern times and passed down from generation to generation between technical workers, and then only after due examination and the exchanging of a secret handshake. It has been known previously only to a select few in each facet of research The railways department cottoned on to it quicker than most when they noticed that a lot of smoke was escaping from their steam locomotives and they switched to diesel. These also leaked a little, and the trend today is toward almost leakproof electric trains.

I suppose that it is only a quirk of nature that the North American Indians did not invent wire as they obviously had figured out that messages could be displayed by controlled leaking of smoke, but did not take that all important step of enclosing it. Sitting Bull could have been the World's First Programmer! □

From an idea originally read in an MG Car Club magazine; author unknown.

Graphics Techniques
- the fundamental concepts

We've all seen computer graphics but wouldn't you like to program your own? Miroslav Kostecki explains the concepts and then shows how . . .

COMPUTER TECHNOLOGY is a tool which extends the abilities of our bodies and minds. With it we can 'build' simulated objects and manipulate them at will by using various computer procedures or algorithms. These techniques, making possible the amazing graphics we have all seen on expensive machines, have recently become available on computers within reach of the home user.

This 'graphics revolution' has been sparked off by a number of developments — first of all, the cost of video hardware is plummeting as a direct result of the falling cost of RAM (Random Access Memory) chips. This has made high resolution graphics more readily available to the average user.

More importantly, however, there is the increasing emphasis on user friendliness and the more natural it feels to use a computer, the greater is its effectiveness as a design and analysis tool. It seems that the more transparent the system, the better results users can get from it.

This software push has in turn caused a rapid development of faster and higher resolution video technology and so the cycle of better hardware and software continues. You can expect that high speed, TV quality scenes will soon appear on your computer monitor — if they haven't already.

Video Display Technology

To gain an understanding of how graphic software procedures work, it's necessary to examine how the computer display system works. The way the computer is wired (the hardware), determines the types of manipulations that are possible with the software.

For example, we must keep in mind that the image is updated for display at a predetermined rate, usually 25 times a second — any slower than this and animated sequences 'flicker.'

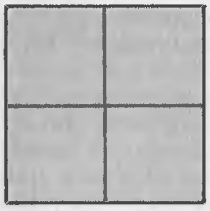
```
10 '      ### DRAW LINE ###
20 ' # Miroslav Kostecki # August 1987 #
30 'draws line from point a,b to c,d
40 'distributed as evenly as possible
50 'using only plots
60 '
70 'use integer variables
80 DEFINT a-z
90 GRAPHICS PEN 1
100 CLS
110 a=100: b=100
120 c=600: d=205
130 GOSUB 180
140 '
150 END
160 '       draw line between points a,b and c,d
170 '
180 u=c-a: v=d-b  'distance along and up
190 d1x=SGN(u): d1y=SGN(v)  'diagonal direction
200 d2x=0: d2y=SGN(v): m=ABS(v): n=ABS(u)
210 IF m<n THEN d2x=SGN(u):d2y=0: m=ABS(u):n=ABS(v)
220 '   m is the larger of abs(u) and abs(v)
230 '   and n is the smaller
240 s=INT(m/2)
250 '
260 FOR i=0 TO m
270  PLOT a,b
280  s=s+n
290  IF s<m THEN a=a+d2x: b=b+d2y: GOTO 320
300  s=s-m
310  a=a+d1x: b=b+d1y
320 NEXT i
330 RETURN
```

Listing 1. *A Draw Program using the algorithm in Figure 3. The program tests to see if it is taking the shortest path from the starting pixel, to the finish.*

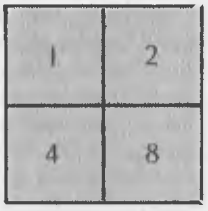
Block Graphics

Early displays were based on blocks of dots forming character symbols — the familiar text screen. This has generally been extended, so that in the full IBM character set, for example, we now have 255 different symbols; these are defined on 8 dot by 8 dot squares and stored in permanent ROM in the video electronics. These symbols include the alphabet (upper and lower case), lines, shapes and blocks of shading.

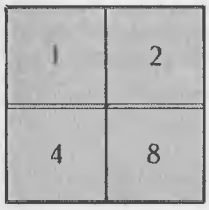
The display screen is usually divided into 80 blocks across and 25 blocks down, giving a possible 2000 characters displayable at once. The graphics systems stores the character codes in RAM (1 byte per code); each of these 2000 bytes can be accessed and changed by computer programs and procedures.



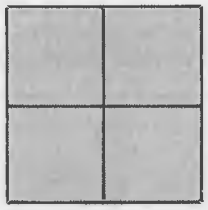
All 4 quarters are blank, so the graphics character code is 128.



Each quadrant is given a value for a 'dot' in that quadrant.



The new character code is 128 (blank) plus the values of the dots —
128 + 2 + 4 + 8 = 142



Here, the character code is —
128 + 2 + 4 = 132

Figure 1. *Block graphics on a text screen.*

A common method of using a text screen to display dot graphics, is to use graphic symbols which are divided into quarters. Each quarter can be either black or white, resulting in 16 possible symbols for each character position.

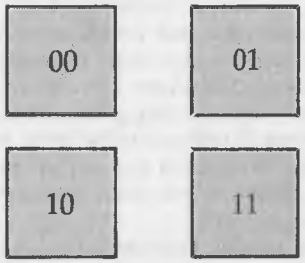
The blank symbol is given a certain set character code (128 in the IBM set) and each quadrant is given increasing values of 1, 2, 4 and 8. When a dot is added to a certain position, the new character code is arrived at by adding the value of the quadrant — see Figure 1.

The use of graphics symbols among text has made possible graphics transfers over modems — this technique is used by videotex systems like Viatel because it is very conservative with respect to the number of bytes needed to create a display.

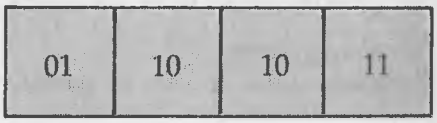
Another advantage over newer types of displays is the quick manipulation of characters on the screen. Only 1 byte in memory must be changed to change a whole character.

Dot Manipulations Onscreen

Graphics is normally displayed on what is called a memory-mapped or graphics screen. Here each dot (pixel) is held in memory and can be accessed and changed separately. This means any pixel can be set on or off at any position of the display which makes the graphics much more flexible. Figure 2 shows how the pixels are stored in groups to form bytes of screen memory.



Four colour pixels, each with its own binary code.



These are stored in the display memory as an 8-bit byte. Each byte is displayed on the screen in sequence to give the 'picture.'

Figure 2. *Pixels on a memory-mapped graphics screen.*

```
10 '          ### FILL ###
20 ' follows the left and right edges
30 ' # Miroslav Kostecki # August 1987 #
40 '
50 MODE 1: DEFINT a-z
60 GRAPHICS PEN 1'border & 10 random lines
70 MOVE 0,0: DRAW 0,398
80 DRAW 639,398: DRAW 639,0: DRAW 0,0
90 FOR i=1 TO 10:DRAW RND*640,RND*400:NEXT
100 '
110 GRAPHICS PEN 2
120 m=2             'size of pixel
130 x=320: y=200    'start position
140 GOSUB 180       'fill x,y
150 '
160 END
170 '      fill subroutene
180 c=TEST(x,y): a=x
190 ud=-2: tx=x: ty=y
200 WHILE a<=x
210  WHILE TEST(a,y)=c: a=a-m: WEND: a=a+m
220  WHILE TEST(x,y)=c: x=x+m: WEND: x=x-m
230  MOVE a,y:DRAW x,y
240  y=y+ud
250  WHILE TEST(x,y)<>c: x=x-m: WEND
260  WHILE TEST(a,y)<>c: a=a+m: WEND
270 WEND
280  IF ud=2 THEN RETURN
290 ud=2: x=tx: y=ty+ud: a=x
300 GOTO 200
```

Listing 2. *A Fill Program illustrating one simple algorithm for coloring a 'block.'*

An extension of the simple memory-mapped screen is a palette system for storing shades and colours. With this system, a number of colours are taken from a particular table and stored in memory.

For example, you might be working in a resolution that allows for only 4 colours on the screen at any given time. If, say, there are 27 colours in the table, you can use all the colours but not at the same time. The 4 colours you choose may be changed at will.

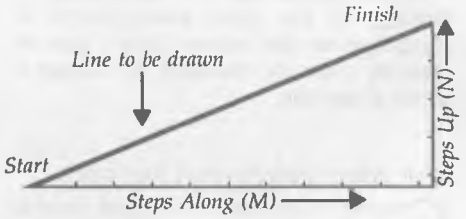
The unlimited manipulation of adjacent pixels on a memory-mapped display system has enabled graphs, spray effects and textures to be applied to the video screen.

Drawing Lines

To draw a line, it must be converted from the continuous form we see on paper into a sequence of discrete pixels that can be displayed on a monitor. Of course, the line's final representation should resemble its intended form, looking like a straight line with consistent width and accurate position.

Early line drawing algorithms used a horizontal and diagonal lines approach. These were easy to implement and were fast but they were also very restrictive. The algorithm now widely used to draw lines is presented in Figure 3. In a nutshell, the process tests to see if it is taking the shortest path from one pixel to the next. Study the diagram and then try the Draw Program in Listing 1 yourself.

If you use low resolution displays, you will have notices 'jaggies' or stair-stepping to the sides of lines. A technique for reducing this effect, called 'anti-aliasing', is to vary the brightness of the line with less bright neighbouring pixels. As a result, lines and edges will look nearly continuous and may even appear to have more precision than the resolution of the screen.



1) Let S be M/2.
2) Move along 1 step.
3) Now, let S be S + N.
4) If S is now greater than M, then
a) Move up 1 step.
b) Let S now be S — M.
5) Loop until the line has moved M steps.

Figure 3. *The algorithm commonly used to draw a line.*

One of the uses for lines used this way is to greatly speed up plotting shapes. As an example, imagine plotting a circle using only discrete pixels — you need to plot points close enough for there to be no gaps. However, if you use lines you need to calculate a point for only every 2 degrees of the circle and then draw lines between them. In most cases, this gives an indistinguishable result but it's many times faster.

Filling Areas

A block of colour can easily be created by drawing layers of horizontal lines but a universal fill algorithm to fill any set shape greatly simplifies programming.

The Fill Program in Listing 2 illustrates one simple algorithm. The left and right edges of the block are found by testing the dots along each direction and a horizontal line is drawn between them. The next line down is then similarly tested and drawn until the left and right edges meet. Then the process is repeated going up. This results in a fast algorithm — but it will not work for concave shapes and may even erase parts of them. Careful placing of the starting position will enable the user to fill a complex shape with few starting points.

More complex fill algorithms are used to fill a shape with a certain pattern. These programs work similarly but refer to a look up table when they apply the points to the screen. Masking can be achieved with single colour fills to obtain areas of textures — first, create a block of texture, then apply an outline in a certain colour; then filling to the edge of the outline colour leaves a textured shape.

I hope these insights into the basics of your computer's graphics system has launched you into graphics processing technology. In the next article, I'll explain how to turbocharge slow graphics. □

Graphics Techniques - Part 2

Graphics tend to be processor intensive which makes them slow by default – here are some techniques to speed things up!

FROM THE TIME of their invention, computers have been designed to be ever faster. But why? To start, eliminating delays can have a powerful effect on programs, especially in the field of graphics. High quality, mathematically generated graphics are generally very processor intensive and, therefore, run very slow. Although, with the new wave of graphics orientated machines, such as the Amiga and Archimedes, speed is less of a problem.

If you don't have the fortune of owning a machine in the hyper-speed category, you realise that most of the current computers running a Basic interpreter are exceedingly slow. This has resulted in a number of techniques being developed in both software and hardware to speed things along.

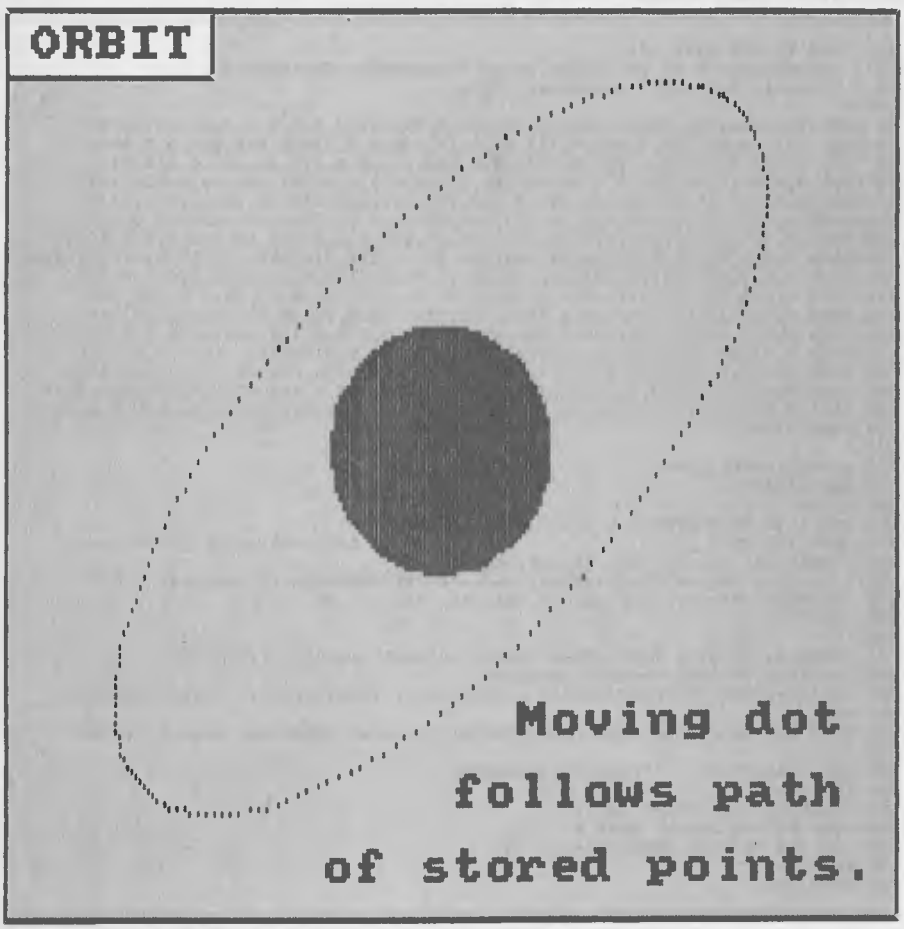


Figure 1. In Orbit — one dot quickly replaces another to give a flicker free image.

Which Language To Use?

Various languages have been developed which compare to Basic in ease of use, but compile into machine code to run at top speed — usually 10 to 100 times faster than Basic. One example is Pascal, which seems to be well suited to graphics manipulation due to its structure. Otherwise, compilers are available for most versions of Basic. (When choosing a compiler make sure it includes a large collection of graphics commands as many do not support graphics at all.)

Alternatively, you may create short sections of machine code and access them from your Basic program. Of course, a knowledge of the workings of your microprocessor is required to perform operations such as this, whereas compilers reduce this need. One good technique is to program a working version completely in Basic and then write machine code for the slower sections, one at a time.

There are two ways to create these machine code programs: if the program is short, you can enter the code straight into Data statements and then Poke these numbers into a portion of memory. The

```
10 ' ### ORBIT ###
20 ' Array storage Demonstration.
30 ' Miroslav Kostecki, Sept. 1987.
40 '
50 MODE 1: GRAPHICS PEN 1
60 DIM s(180), c(180)
70 DIM ax(180), ay(180)
80 '
90 DEG              'Store every 2 degrees
100 FOR dg=0 TO 180
110  s(dg)=SIN(dg+dg):c(dg)=COS(dg+dg)
120 NEXT dg
130 '
140 ox=320: oy=200          'Draw circle
150 size=160: ss=size/3
160 FOR dg=5 TO 180 STEP 5
170  MOVE c(dg-5)*ss+ox, s(dg-5)*ss+oy
180  DRAW c(dg)*ss+ox, s(dg)*ss+oy
190 NEXT dg
200 '
210 FOR dg=0 TO 180         'Store points
220  x=c(dg)*size+ox
230  y=s((dg+68) MOD 180)*size+oy
240  PLOT x,y
250  ax(dg)=x: ay(dg)=y
260 NEXT dg
270 '                       Move dot around
280 FOR dg=1 TO 180
290  PLOT ax(dg-1),ay(dg-1),0
300  PLOT ax(dg),ay(dg),1
310 NEXT dg
320 GOTO 280
```

Listing 1. Orbit gives a demonstration of array storage.

subroutine (CALL), can then be accessed directly from the Basic program when required.

The second option involves using a compiler or assembler to generate the machine code and save it separately. When the routine is wanted at a later period, it's loaded off the disk into the required memory area.

The examples presented throughout this series will be in Basic as the majority of computer users understand this language, and most computers come supplied with it as standard. However, it is the logic behind the program, the algorithm, which is of real interest. The demonstration programs will be simple and therefore easy to convert to other languages.

An Animated Concept

Generating screens while a program is running tends to be a long and tedious process. To counteract this, single screens of graphics can be stored and recalled later. However, real time movements, updated and animated screens must reach a certain speed to be useful. For example, when rotating a 3D model the calculations and complex screen manipulations must be made within a certain time period.

Essentially, animation gives the illusion of an object being 'alive'. Various procedures have been developed for this concept, the most common method is to draw an object in one position, calculate its next position and then erase the object and draw it in the new position. That's very similar to drawing pictures on the corner of a note pad and flicking through them to give the impression of movement.

However, there's the problem of a delay between erasing an object and redrawing it. This causes a blank period and introduces flicker. Ideally, the object should be erased and redrawn in the same instant of time, eliminating the blank period.

Another potential problem with flicker, is the fact that to make the movement appear smooth, each frame must be updated faster than the eye can detect. This flicker free speed is around 20 frames per second. Thus, each frame should last for no more than 1/20 th of a second. Although an acceptable speed may be slower, this speed can be readily achieved on small computers by applying certain techniques.

Tables and Arrays

Tables and arrays are used extensively in computers. Any group or sequence of numbers that must be easy to access either sequentially, or randomly, can be stored in an array. Computer graphics sys-

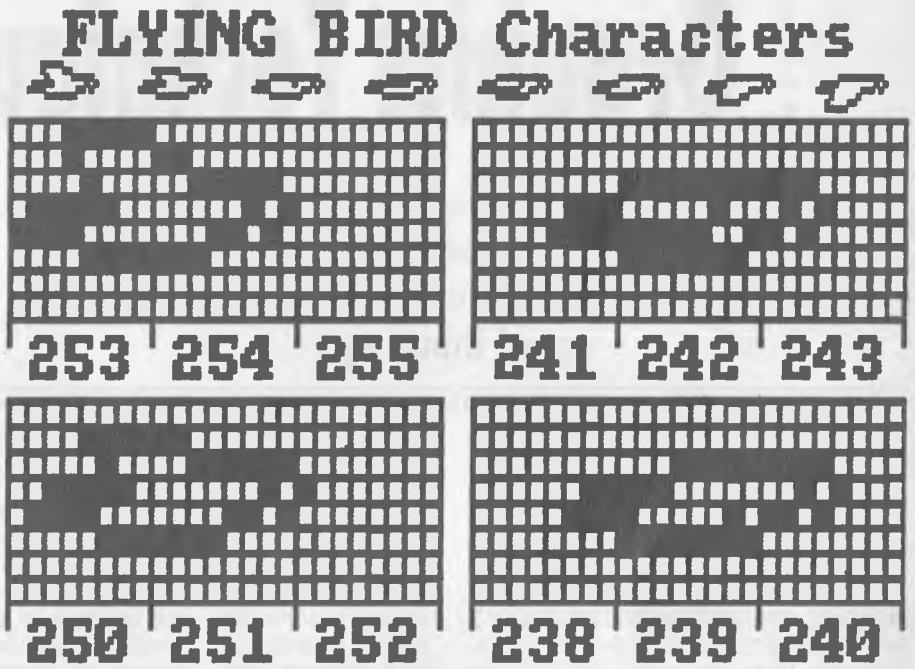


Figure 2. Flying Bird — the wings and body move one pixel at a time, so that the animation appears smooth.

```
10 ' ### FLYING BIRD ###
20 ' Demonstration of Animation using Programable Characters.
30 ' Miroslav Kostecki, September 1987.
40 '
50 DATA 31,16,8,124,240,15,0,0,0,192,62,5,26,224,0,0,0,0,0,0,0,0,0,943
60 DATA 0,15,4,62,120,7,0,0,0,192,63,2,13,240,0,0,0,0,0,128,0,0,0,0,846
70 DATA 0,0,3,28,60,3,0,0,0,0,255,17,6,248,0,0,0,0,128,64,128,0,0,0,940
80 DATA 0,0,0,15,30,1,0,0,0,0,255,248,3,252,0,0,0,0,192,160,64,0,0,0,1220
90 DATA 0,0,0,7,15,0,0,0,0,0,255,4,249,254,0,0,0,0,224,80,160,0,0,0,1248
100 DATA 0,0,0,3,7,0,0,0,0,0,31,224,130,255,0,0,0,0,240,40,208,0,0,0,1138
110 DATA 0,0,0,1,3,0,0,0,0,0,31,224,192,67,124,0,0,0,248,20,232,0,0,0,1142
120 DATA 0,0,0,0,1,0,0,0,0,0,15,240,224,32,67,124,0,0,252,10,116,128,0,0,1209
130 DATA 0,0,7,120,240,16,33,62,0,0,254,5,58,64,128,0,0,0,0,0,0,0,0,0,987
140 DATA 0,0,3,60,120,8,15,0,0,0,255,2,29,96,128,0,0,0,0,128,0,0,0,0,844
150 DATA 0,0,0,31,60,7,0,0,0,0,255,1,22,248,0,0,0,0,128,64,128,0,0,0,944
160 DATA 0,0,1,14,31,1,0,0,0,0,255,8,243,252,0,0,0,0,192,160,64,0,0,0,1221
170 DATA 0,0,0,7,15,0,0,0,0,0,127,252,1,254,0,0,0,0,224,80,160,0,0,0,1120
180 DATA 0,0,0,3,7,0,0,0,0,0,127,130,128,127,0,0,0,0,240,40,208,0,0,0,1010
190 DATA 0,0,0,1,3,0,0,0,0,0,126,33,240,192,63,0,0,0,0,248,20,104,128,0,0,1158
200 DATA 0,0,0,0,1,0,0,0,62,33,16,248,224,31,0,0,0,128,124,10,52,192,0,0,1121
210 DATA 17091
220 '
230 SYMBOL AFTER 208
240 DIM b$(16)
250 tt=0
260 FOR i=45 TO 0 STEP -3: t=0
270  FOR j=0 TO 2                          'Read data and setup characters
280   READ a1, a2, a3, a4, a5, a6, a7, a8
290   t=t +a1 +a2 +a3 +a4 +a5 +a6 +a7 +a8 '<<<checksum for errors
300   SYMBOL 208+i+j, a1, a2, a3, a4, a5, a6, a7, a8
310  NEXT j
320 '
330  READ s: IF s<>t THEN PRINT "Error in Data line";16-i/3: STOP
340  tt=tt+t '<<<2nd overall checksum
350  b$(16-i/3)=" " +CHR$(208+i) +CHR$(209+i) +CHR$(210+i)   'store groups
360 NEXT i
370 READ ss: IF ss<>tt THEN PRINT "Error in Data- check for double copies"
380 '
390 j=1: delay=20     ' Loop for movement
400 FOR i=1 TO 16
410  LOCATE j,10: PRINT b$(i);
420  FOR d=1 TO delay: NEXT d
430  IF i=8 OR i=16 THEN j=j+1
440 NEXT i
450 GOTO 400
```

***Listing 2.** Flying Bird demonstrates animation using programmable characters.*

tems make use of arrays to store representations of objects, to store sequences of movements, and many other attributes of the graphics screen, with its movement and calculations. This flexibility quickens execution of graphics and makes the program much easier to follow.

Storing numbers in a two dimensional array to represent objects on the screen has been standard practice for many years. Objects can be represented in an array where the program only tests and manipulates the array. In turn, a graphics program would use the array to produce the graphics on screen. A simple example would be to store either 0, 1 or 2 in a 3 x 3 array to represent a blank space, or a O or X in a game of tic-tac-toe. Similarly, image descriptions are stored for easy manipulation of 3D objects for rotation and movement.

Into Orbit

The Orbit program in Listing 1 reveals some of the routes arrays can take to speed up both the generating of still graphics and animated sequences.

Sines and cosines as well as any other functions can be stored in arrays for later access, especially if they take a long time to calculate. In this program we store the sine and cosine value for every degrees.

Shapes are much more quickly drawn using short lines between points so that only these particular points need to be calculated. In the program, a circle is drawn using lines from the preceding point to the current point in steps of 10 degrees.

If points are to be accessed frequently, store the points in an array for later use. The program also plots the stored points to show its workings more clearly.

The last section is the actual loop which moves the dot from point to point. The technique is to wipe out the last dot and plot the current dot as quickly as possible to avoid flicker due to the blank period.

Figure 1 shows a graphic printout of the process where one dot is quickly replaced by another to give a flicker free image. The points plotted here are principally blank except for one particular ever-changing pixel which gives the impression of moving around the circle by switching from one point to the next.

Programmable Characters

On the majority or graphics systems, you can program the individual characters which make up the text screen by defining the dots that the character consists off. Systems such as these are called programmable graphics, or symbols. Various alphanumeric styles and

In a different mode . . .

```
Rem NB include "Def fnoffset(x,y)=160 * y + (x Or 1)"

Sub Pdraw(zx,zy,dotcolor) Static
Rem draw a point
  dotnow = Peek(fnoffset(zx,zy))
  If zx Mod 2 <> 0 _
        Then   dotclr = (dotnow And &hf0) Or dotcolor _
        Else   dotclr = (dotnow And &hf) Or (dotcolor * 16)
  Poke fnoffset(zx,zy),dotclr
End Sub

Sub Ldraw(x1,y1,x2,y2,dotcolor) Static
Rem draw a line
  deltax = x2-x1:deltay=y2-y1
  If deltax<>0 Or deltay<>0 Goto Hline
  Call Pdraw(x1,y1,dotcolor)
  Exit Sub
Hline:
  If deltay<>0 Goto Vline
  For zx=x1 To x2 Step Sgn(deltax)
        Call Pdraw(zx,y1,dotcolor)
  next zx
  exit sub

Vline:
  If deltax<>0 Goto DPline
  For zy=y1 To y2 Step Sgn(deltay)
        Call Pdraw(x1,zy,dotcolor)
  Next zy
  Exit Sub

DPline:
  If Abs(deltay) < Abs(deltax) Goto DMline
  slope = deltax / deltay
  For zy=y1 To y2 Step Sgn(deltay)
        zx = slope * (zy-y1) + x1
        Call Pdraw(zx,zy,dotcolor)
  Next zy
  Exit Sub
DMline:
  slope = deltay / deltax
  For zx = x1 To x2 Step Sgn(deltax)
        zy = slope * (zx-x1) + y1
        Call Pdraw(zx,zy,dotcolor)
  Next zx
End Sub
Sub Setup Static
rem set-up for lo-res graphics
  Screen 0:Width 80
  Key Off:Cls
  Out &h3D8,9
  a=&h3D4:d=&h3D5
  Out a,4:Out d,&h7F
  Out a,6:Out d,&h64
  Out a,7:Out d,&h70
  Out a,9:Out d,1
  Def Seg = &hB800
  For c=0 To &h3FFE Step 2
        Poke c,&hDE:Poke c+1,0
  Next c
End Sub
Sub Rest Static
rem re-set to 80x25 text
  a=&h3D4:d=&h3D5
  Out a,4:Out d,&h1F
  Out a,6:Out d,&h19
  Out a,7:Out d,&h1C
  Out a,9:Out d,7
  Cls
End Sub
```

Listing 1. *Low resolution graphics for the IBM PC/XT.*

If you're feeling more adventurous, try these routines for low resolution graphics on the IBM PC/XT, compliments of Jeff Richards.

THE IBM PC XT CGA has a low resolution graphics mode that is not implemented in the BIOS and not documented. Using this mode involves resetting some of the screen controller's registers and doing direct screen access to control the pixels. Resolution in this mode is 160 x 100 with 16 colors (although 1 colour is black, and another is bright black!).

Three routines are presented here. The first is a collection of QuickBASIC subprograms to draw pixels and lines. the second and third are two programs that use the pixel-drawing subprogram to demonstrate the new mode. Scatter draws a display like a kaleidoscope. Lines creates 16 projectiles (2 are invisible) and bounces them around the screen leaving a coloured trail.

Note that the initialization subprogram (Setup) must be called before plotting any points, and the de-initialization subprogram (Rest) must be called before returning to the operating system.

The subprograms first appeared in a form for BASICA in *Dr Dobbs Journal* - Number 84, October 1983. To make life easier (for those of you with a modem), these listings are on YC's Bulletin Board).

```
Defint a-z
Def fnoffset(x,y)=160 * y + (x Or 1)
Dim dotc(15), xpos(15), ypos(15), cdir(15)

     Call Setup
     For i=0 to 15
       dotc(i) = i
       xpos(i) = 80
       ypos(i) = 50
       cdir(i) = Int(Rnd*4)+1
     Next i
     While Inkey$ = ""
       For i=0 to 15
         If rnd<.2 Then cdir(i) = Int(Rnd*4)+1
         xinc=1:If cdir(i)>2 Then xinc=-1
         yinc=1:If cdir(i) Mod 2 = 0 Then yinc=-1
         xpos(i)=xpos(i)+xinc:ypos(i)=ypos(i)+yinc
         If xpos(i)<0 Then xpos(i)=0:cdir(i)=cdir(i)+2 _
          Else If xpos(i)>159 Then xpos(i)=159:cdir(i)=cdir(i)-2
         If ypos(i)<0 Then ypos(i)=0:cdir(i)=cdir(i)-1 _
          Else If ypos(i)>99 Then ypos(i)=99:cdir(i)=cdir(i)+1
         Call Pdraw(xpos(i),ypos(i),dotc(i))
       Next i
     Wend
     Call Rest
     System
```

Listing 2. Lines — a pixel drawing program to demonstrate the undocumented low resolution graphics mode.

```
Defint a-z
Def fnoffset(x,y)=160 * y + (x or 1)

     Call Setup
     While Inkey$ = ""
       dotcolor = int(rnd*16)
       n = Int(Rnd*50) :m=Int(Rnd*50)
       Call Pdraw(80-n,50-m,dotcolor)
       Call Pdraw(80-n,50+m,dotcolor)
       Call Pdraw(80+n,50-m,dotcolor)
       Call Pdraw(80+n,50+m,dotcolor)
       Call Pdraw(80-m,50-n,dotcolor)
       Call Pdraw(80-m,50+n,dotcolor)
       Call Pdraw(80+m,50-n,dotcolor)
       Call Pdraw(80+m,50+n,dotcolor)
     Wend
     Call Rest
     System
```

Listing 3. Scatter — this program will give a kaleidoscope display with 16 lines bouncing around the screen leaving coloured trails.

graphics symbols can be created with this method, and dedicated editors exist which help the process along.

Programmable characters have the superiority of being faster and simpler than plotting and drawing shapes, characters and graphics. However, as with most things in life, there are disadvantages: for example, they have to be located within a particular character position. This can be overcome on some computers by fixing the graphics system so that it prints characters onto the graphics screen in any dot position.

As a general rule, programmable characters can be displayed in only one shade or colour. By setting the background as transparent, though, some systems can overlap many layers to produce multiple coloured character blocks. On some graphics machines, such as the Commodore 64, automatic sprites are even possible. They are similar to programmable characters but contain a higher degree of flexibility. After specifying the direction, speed and distance, the computer moves the sprite without erasing the background. Automatic collision detection between sprites also exists.

In most cases, the characters are stored in strings and printed onto the screen whenever their services are required. Control Characters to locate these symbols anywhere on the screen are generally included. Animation is achieved by displaying one character and then switching to the next one, and the next, and so on. Movements can be generated by switching between transparencies.

Flying Bird Program

In Figure 2, you can see what has been done to the basic bird character... it has been moved forward step by step with its wings moving one pixel along each time, so that the animation appears smooth. It's not necessary to keep a character rigid.

The first thing you notice about the program is the heavy use of data (which is check-summed to reduce errors). The data is read and converted into symbols or programmable characters. The commands and technique differs slightly from machine to machine.

The characters are then stored in an array following the order that they are shown in.

The method of display here is: move to one location, display eight frames, move to the next location and display the next eight frames, and so on. This culminates in the creation of a life-like flying bird.

Next, we'll examine more sophisticated techniques to speed up animation. □

Graphics Techniques - Part 3

Even with the techniques in Part 2, computer graphics can still be slow – so let's look at more sophisticated paths to speed . . .

PICTURE THIS: You have finally completed a brilliant graphics program which is your one way ticket to wealth. But, of course there is just one last problem — the program runs too slow. After carefully examining the situation, you find that the movements of data around the screen memory are holding up the system. However, this computer cannot push data around any faster. It's already at top speed, it seems.

If this hasn't happened to you yet, it soon will. So, is there anything a programmer can do without using scissors and pliers on the wiring? Believe it or not, most computers already have inbuilt devices to solve this problem.

Block movement

However, before we examine these devices, lets take a look at what a 'block movement' is. Put simply, a block movement is the transfer of a large number of data bytes or words. These transfers can be made within the main memory storage or between another device on the outside. This type of mass movement is made within your computer very often.

As an example, one common block movement is storage of the graphics screen memory onto a disk. This process lets you take your time to generate the graphics but allows quick access when it's needed. Your computer's instruction manual will contain the necessary commands. Of course, your screen memory may be stored in a different section of the main memory instead of the disk.

A more useful block movement saves only a small section of the screen at a time. These rectangular areas of the screen are customarily called graphics windows. Because much less data needs to be moved, each block move is much faster. Small graphics windows also permit you to store many more windows in your memory space.

The remarkable versatility and speed of the graphics window system has made a big impact in animating computer graphics. Split second frames for animation are created and stored beforehand. Then the frames are recalled at high speed to create an animated scene — see Figure 1. Of course, the animation does not have to be recalled to the same place as it was generated. The location may even be made to move around while the animation is happening.

Great, now that we know what block moves are, what about our original problem? What can we do to speed up these transfers?

DMA controllers

The majority of computers have dedicated brute 'servants' to do their block transfers for them while they get on with the complicated calculations. These Direct Memory Access controllers (DMAs) really do have minds of their own but are much less intelligent than their master, the Central Processing Unit (CPU).

The CPU will leave messages telling the DMA which block moves to make along with other special instructions. It passes these commands by leaving messages in the DMAs registers. Registers are in effect, just message boxes.

A DMA will do the actual work of moving memory sections with little disturbance to the CPU. This means the CPU can be making calculations instead, so, overall, the program will run much faster. In fact, on most systems, a DMA can transfer data a great many times faster than the CPU could anyway.

Recently developed computers have a device (Bit Blitter) similar to a small army of DMAs under the control of their own intelligent controller device. These Bit Blitters are on most occasions connected directly to the graphics screen memory for incredibly fast block movements. The Commodore Amiga and Atari Mega ST owe a large portion of their graphics power to the Bit Blitter chips.

Check if your computer has a DMA device or a Bit Blitter. To use the device you will need to find out about its register structure and the commands to use. The time you spend on research will be well worth it.

So far we have assumed that we must literally move the data from one position of the graphics screen to the next. In fact, what we really want is to display the block of data at a different position on the screen. If we could change the position of display, the graphics data can stay where it is. This leads us into examining the video display system more closely.

The CRT controller

At the heart of any video display system is a device called a CRT Controller. This is another 'slave' tool which is controlled by a register system, like the DMA units described above. However, this device has the job of converting the screen memory into video signals for the display monitor. And like most computer circuits today, these CRTCs are designed to be remarkably flexible.

The chip will construct its video signals in the way specified by its registers. Changing one of these registers will alter the display on your monitor without any of the screen memory being moved at all. Therefore, this change can happen within millionths of a second.

One or two of the registers will control the scrolling of the display, either up, down, left or right one pixel or character.

Another register may point to the screen memory area. When a second display is needed, all you need to do is change the pointer. In fact you might switch it between many screen memory areas.

Usually, when extra screens are set up, programs are working on a screen which is not actually being displayed at the time. A switch is made to this screen only when it is totally completed, giving the user an instantaneous, new display.

Palette switching

Recent video control systems, and sometimes CRT Controllers themselves, contain something which deserves a section of its own — a Palette Switching system.

The range of available colours on most computer systems is much greater than the number that can be displayed at any one time. This complete range of colours is called the 'Colour Palette'. You must assign a colour from this palette to each of your 'pens'.

For example, the palette may contain 512 colours, while the video display can handle only 4 at a time. So each of the 4 pens is given one of 512 colours from the palette. The next stage involves the storing of the 4 colours by the Graphics Display System so that the pens appear on the screen in their selected colours.

The real power of this system is that by simply changing the colour selection, all of a particular pen on display will change colour instantly. At first, the usefulness of changing colours seems to be limited to just flashing between different colours. However, some interesting changes happen when you start to select the same colour for different pens.

As an example, take a look at the UFO program in Listing 1. First, an oval is drawn at 45 degrees on the screen. Spokes are then drawn through the center of the oval in 13 different pens, each a different colour. The screen has now been set up for palette switching.

All 13 pens are now set to the background colour, thus turning them invisible. Pen 1 is then set to white, causing it to be the only visible spoke in the oval wheel. To create the appearance of the

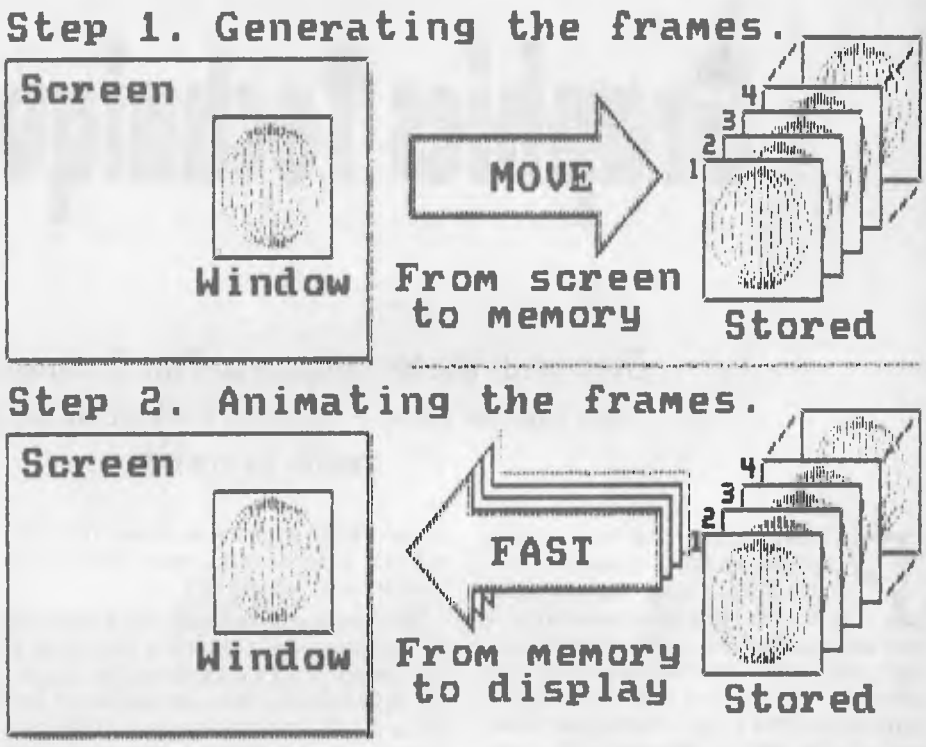


Figure 1. Split second frames for animation are generated and then moved to memory; the image is animated by recalling the frames from memory at high speed.

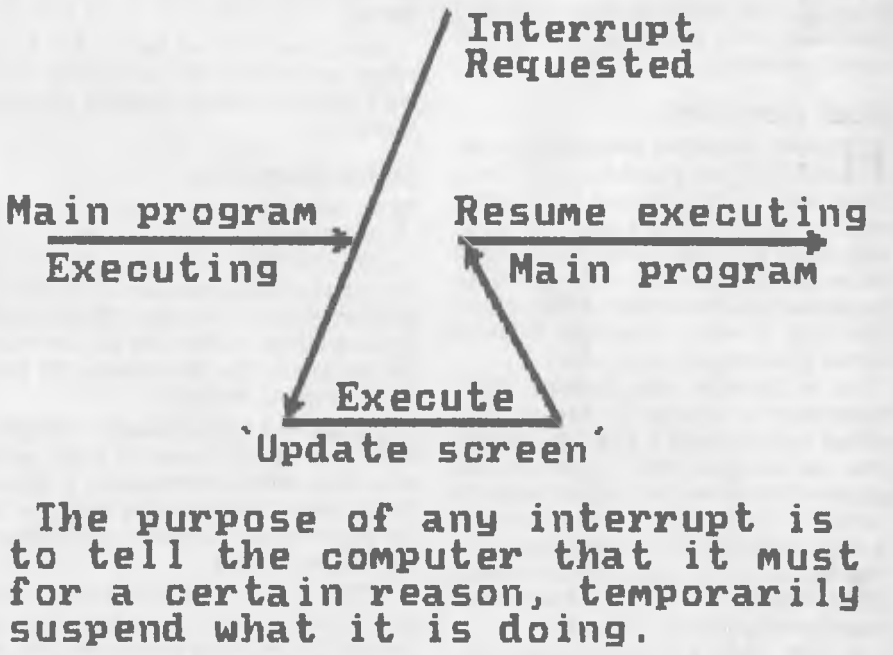


Figure 2. The purpose of the interrupt might be to make another scan of the keyboard, bring about an update of the screen or just to tell the CPU that the video has finished a screen frame. After dealing with this interruption, the central processor will continue where it left off.

spoke transferring to the next position is quite easy. We simply return the pen to the background colour and set pen 2 to white. This redirection of colour is almost instantaneous. Note, though, that none of the pixels on the screen have actually moved, only the colours have changed. By repeatedly moving from one spoke to the next in this way the line can be forced to rotate at fantastic speeds. Try the program yourself and increase the delay to study the effect more closely.

```
10 ' ££££  UFO  ££££
20 ' Palette switching demonstration
30 ' Miroslav Kostecki. August 1987.
40 '
50 ' setup 16 colours
60 ' out of 27 on 640x400 screen
70 MODE 0
80 DIM s(210), c(210)
90 s1=300: s2=s1/2
100 e=360/13/16
110 ' precalculate points along oval
120 k=1: DEG
130 FOR a=0 TO 360+e+e STEP e
140  c(k)=s1*CDS(a)+320
150  s(k)=s2*SIN(a)+c(k)/4+120
160  k=k+1
170 NEXT a
180 GOSUB 400
190 MOVE 320,200: FILL 15
200 INK 15,10: INK 14,18
210 FOR a=1 TO 104
220  GRAPHICS PEN (a MOD 13)+1
230  MOVE c(a),s(a)
240  DRAW c(a+104),s(a+104)
250 NEXT a
260 GOSUB 400
270 INK 0,0: BORDER 0
280 '
290 'rotate the bright colour
300 delay=40
310 FOR i=2 TO 13
320  INK i-1,10: INK i,26
330   FOR d=1 TO delay: NEXT d
340 NEXT i
350 INK 13,10: INK 1,26
360  FOR d=1 TO delay: NEXT d
370 GOTO 310
380 '
390 ' draw outside of oval
400 GRAPHICS PEN 14
410 FOR a=1 TO 208
420  MOVE c(a),s(a)
430  DRAW c(a+1),s(a+1)
440 NEXT a
450 RETURN
```

Listing 1. The UFO program.

A similar method uses groups of pens which are turned on at the same time. By 'turning off' the last pen and switching on the next one in the row, the whole group will seem to move slightly forwards. This might leave half the pens 'on' and half 'off'. Notice, though, how much smoother the movements are when small parts are changed.

Animation using pallete switching doesn't end here. Think of the even more incredible animations that can be created by utilizing shapes instead of lines. However, beware! The speed of palette switching is much too fast for most computers. So fast in fact, that delay loops will become almost a necessity in the majority of your programs.

Normally, a computer reads each portion of memory by accessing its own unique address, like a different telephone number for each house on your street. A few years age, most 8-bit computers ran out of any new addresses when they reached the 64 kilobyte mark. This is accounted for by the fact that the total number of combinations is 65,536 for these machines.

Most new computers bypass this limit, or the new 640 Kbyte limit set by an IBM PC, by using Bank Switching. The idea is similar to having an STD area code in front of a telephone number. Imagine having, not one, but four 64 Kbyte 'banks' of memory on your 8-bit computer. Each of these banks can be accessed by changing the bank number (1-4) first. This gives a total 256 Kbyte of usable memory.

Of course, the obvious advantage is that more graphics screens and windows can be stored. However, the addresses stay the same for each bank, so most video control systems will display the same area within a different bank. This means that one screen memory block can be transferred to the next by simply changing the bank.

This system is simple, but gives you headaches when you try writing programs for the system. The programs tend to switch around with the banks. If possible, manipulate the CRT controller instead, as described above.

Check the memory map on your computer to make sure it does switch banks. Then, if you plan to use the system, get as much information as you can about the switching and how it affects your video controller.

Now that our graphics is switching and moving fast enough, how do we control them efficiently?

Interrupts

While the CPU in your computer is at work, a timer or a similar device will periodically send an emergency line to the processor. Usually, the type of emergency is also sent. Immediately, the processor will temporarily leave its current program and attend to the interrupt. The purpose of the interrupt might be to make another scan of the keyboard, bring about an update of the screen or just to tell the CPU that the video has finished a screen frame. After dealing with this interruption, the central processor will continue where it left off — see Figure 2.

These interrupts remain among the most powerful instruments in computer programming. One reason is that, on the whole, a computer will run faster because routine checking is done automatically. Another reason is that programs tend to be much simpler and thus, less prone to errors.

For example, an interrupt may alert a program that a graphics character has reached the edge of the screen. This means that the main procedure can continue moving the character without continually testing for the edges. Another use is to interrupt a program when a 'blank' occurs in the video signal. The screen can then be updated with no visible flicker or tearing.

Most computer programming languages support the use of interrupts. Newer strains of Basic even allow timed interrupts and interrupts to control the sound system. Check the index of your programming manual.

However, a word of warning, the massive use of interrupts will restrict the speed of the main program. If no care is taken, you might find that the computer is only processing interrupts, with no time left for the main procedure. Your computer will then 'hang' and never finish its program.

By now your graphics are zooming around the screen. What you need is some techniques to produce better graphics. Next time I will show you some acrobatics with line graphics as they reflect, enlarge and rotate. □

Graphics Techniques - Part 4

Now that you're familiar with the basic techniques of computer graphics, animation, memory use and interrupts, it's time to write your own CAD system!

WHEN CONSIDERING graphics on a computer, we tend to think of stick figures, line diagrams and shapes. Although much more realistic techniques exist today, two dimensional line drawings are still the most common form of graphics used in business and industry. It's no wonder then that thin line grids and wire mesh designs have been the hi-tech look for many years. There is one type of program, in particular, which promotes this look.

Have you ever wanted to use the computer to design furniture or electronic circuitry and automatically receive the plans from it? Computer programs created to perform functions such as this do exist, ready to use, but at a price!

They are called Computer aided design or Cad programs. Today, many manufacturers simply would not be in business if left suddenly without their use. Units such as buildings, machines, maps and even computers are all designed with Cad.

The point is, all Cad programs rely on interconnecting line techniques a great deal. In fact, it's these techniques and procedures which perform all the calculating work of zooming into maps, rotating machine cogs or moving components around a circuit board.

Flexibility is the key to this type of work, so most graphic design programs are extremely versatile. By concealing complicated logic behind simple hand movements, you can create almost anything, nearly without thinking. This makes testing new ideas almost fun (a dirty word at respectable businesses)!

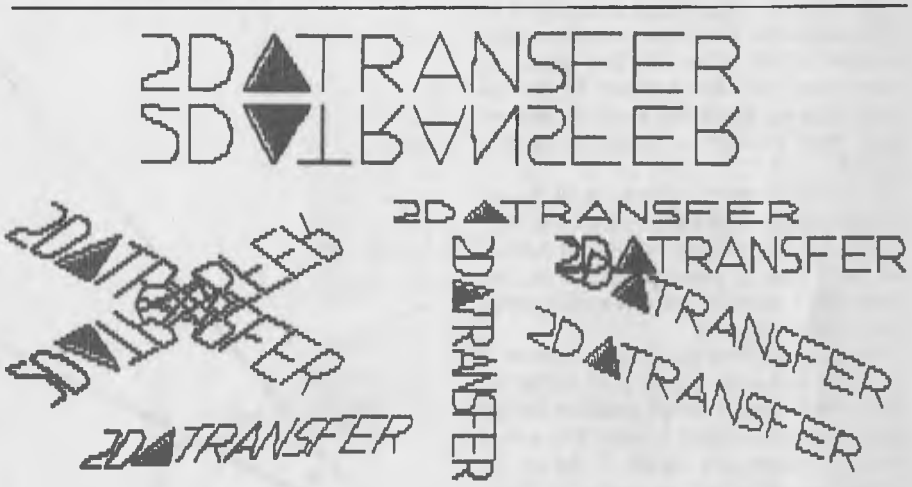


Figure 1. A printout of '2D Transfer' in various reflections, rotations, enlargements and stretchings.

For more information on Cad and similar graphics programs, see the feature articles in the November 1987 issue of *Your Computer*.

How it's done

Of course, there is not enough space here to discuss all the details of Cad. Many large books have been published on the subject and these are by no means exhaustive. However, by examining the way in which some simple transfers are executed, we can obtain a good overall idea of how these work.

The sample printout of 2D Transfer (see Figure 1) in various reflections, rotations, enlargements and stretchings, will give you some idea of what can be done even with a simple program. Try the program on your computer first, then carefully follow Listing I as we explain the various parts.

First you must understand how the shapes and figures are stored within the computer. This is essential if you want to understand the changes and movements because this stored information is what really changes. The display program simply exhibits what is in storage.

Think of the way you draw up a shape in line graphics using the Draw command in Basic. A starting point is selected. Then a line is drawn to the first point. From here, a second line continues to a second point,

a third point and so on until the shape is drawn. The only information needed to draw the shape are the points at all the corners. The lines between the points are drawn automatically. It is these points that we need to store to draw the shape again.

Our program stacks the points by first storing the number of points in the shape. Then the starting point is given, followed by the other points. Of course, they are all pairs of x,y co-ordinates. The succeeding shapes are similarly stored.

The program uses separate Data statements for these shapes. Notice that an end is reached when zero is found for the number of co-ordinates in a set.

Where do we encrypt all this data? The Read Data section will store all the x co-ordinates in the a() array, the y co-ordinates in the b() array and the number of points in each shape are amassed in the s() array.

To draw our shapes, the subroutine at line 780 is used. This works as follows: the number of points in the first shape is taken from the first number in the s() array. Starting at the first point stored, we draw this number of lines using the points.

The next number is taken out of the s() array and the next shape is drawn until the number in the s() array is zero. Of course, we keep track of where we are in the s() array (the i variable) and the a(),b() arrays (using the j variable).

Notice that when the shape is drawn, a constant value of x and y is added to them. This is the starting point or 'offset' amount and is added to shift the actual position where the shape is drawn. By changing x and y we can make the shape appear anywhere we desire.

The position is modified in the subroutine at line 900. To enter this section, press the shift key and move the shape using the cursor keys. Figure 2 shows how this offset point shifts the shapes position.

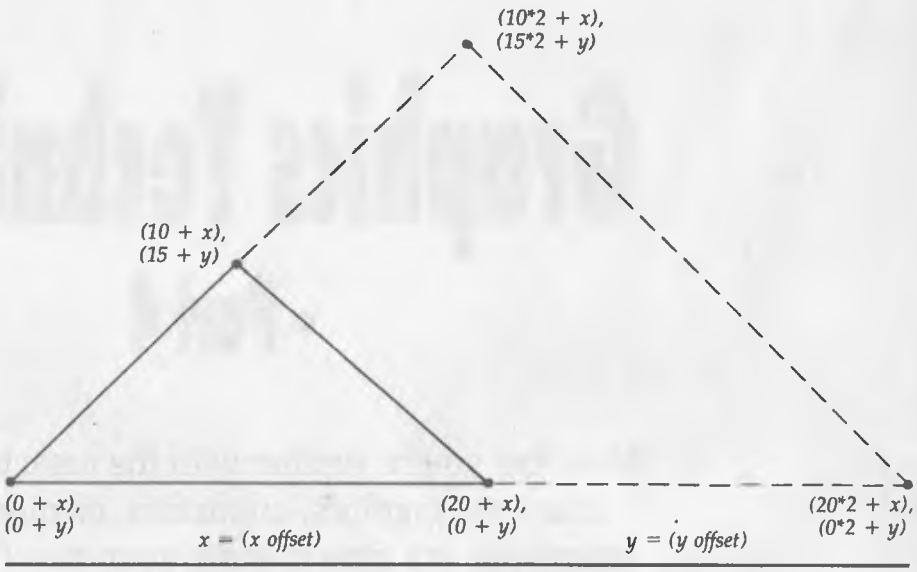


Figure 2. How an offset point shifts the shape's position (see line 900 in Listing 1).

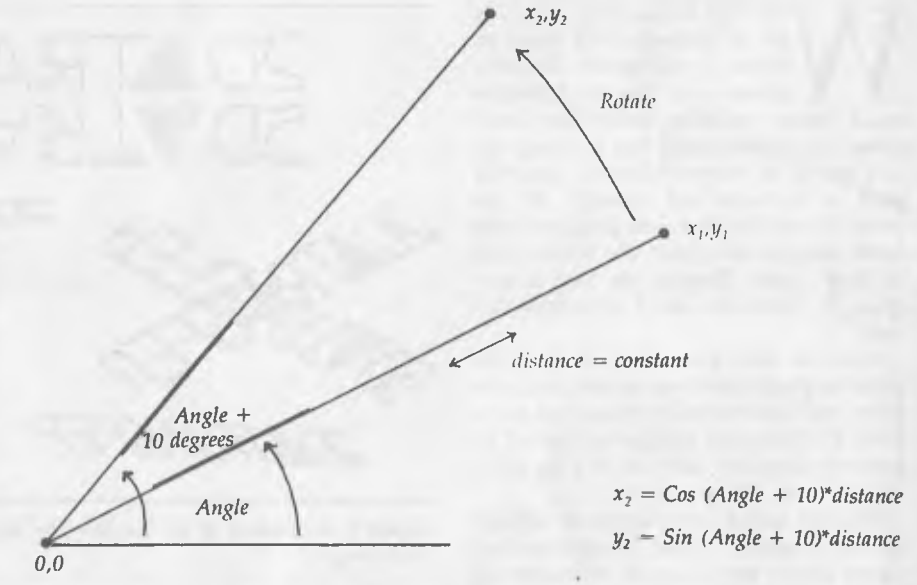


Figure 3. This shows what the equations in line 740 of Listing 1 achieve.

Manipulating the point

Now, because the stored points start at or near 0,0 the manipulations are really very simple. For example, to enlarge or shrink the shape we multiply every point in the arrays a() and b() by a certain factor. To double the size, simply multiply them by 2. This shrinking and enlarging is done with the short subroutine starting at line 620.

Compresses and stretches are even easier: just scale the variable you want in the direction you need. For example, to compress the word to half its length, just multiply all the a() arrays by half. The tiny subroutine designed to do this is found at line 580. Again take a look at Figure 2 to see a visual representation of how this is accomplished.

Another manipulation is the reflecting of shapes up, down, left and right. At first you might think that a manipulation such as this would be very complicated. Actually, it is the easiest manipulation. To reflect in the x axis we make the whole a() array negative and similarly for the y axis. These two subroutines are located at lines 500 and 540.

Now for the difficult part: rotations. The 'rotate by an angle' subroutine starts at line 680 and takes some basic mathematics to understand it. The variable 'l' is the distance from position 0,0 to the point; the equations at lines 720 and 730 work out this distance. It distance must stay the same if we rotate around the point 0,0. Only the angle, as represented by the variable 'ag' changes; -10 or +10 degrees in this program.

Now the new point can be worked out by using the fact that for a line from 0,0; x is the cosine of the angle multiplied by the distance and y is the sine of the angle

multiplied by the distance. These equations are used in line 740. The variable 'q' is the old angle and 'ag' is the amount we want to change it by. So we end up landing our new point at a new angle, but the same distance around 0,0. To rotate the whole series of characters, every point must undergo this process.

If that was over your head, try looking at Figure 3 as you read the explanation again.

To speed things up

Now that you've tried the program, you will have noticed that the rotation takes a good deal more time than the other transfers. This is due to the more complicated number crunching that is involved. To speed up the process, you may like to try storing all the sines, cosines and tangents in arrays and then use these arrays instead of calculating the functions every time.

Another way of speeding the process is to store the lengths to each point from 0,0 and also the angles at which the points are positioned.

This eliminates the need to work these out at all. By storing points like this, rotations and enlargements will be calculated much faster. However, the subroutine which draws the shapes will need to be more complicated and so much slower.

Of course, the faster the computer the faster the manipulations. The resolution too has the usual effect?8a higher resolution is much clearer and more accurate, but it is quite a bit slower because of the greater amounts of data that must be manipulated.

If you work with this faster breed of computer, maybe you are already familiar with these techniques. The latest computers are using these new graphics concepts to add great flexibility to character presentations and the way in which they can be manipulated. Recent computers such as the Archimedes by Acorn even let you redefine the standard text characters in this way. I think we will see more of this type of flexibility the future. □

```
10 '  £££ 2D LINE TRANSFERS £££
20 ' Miroslav Kostecki. Nov, 1987.
30 '
40 MODE 1: INK 1,0: INK 2,13
50 DIM a(100), b(100), s(20)
60 '
70 ' 2D/\TRANSFER
80 DATA 7 ,0,48,29,48,36,38,36,22,31,19,0,9,0,0
,36,1
90 DATA 7 ,46,1,46,48,69,48,84,39,87,23,84,10,71
,0,47,0
100 DATA 3 ,98,0,129,48,161,0,98,0
110 DATA 3 ,176,0,177,48,152,48,202,48
120 DATA 8 ,214,0,215,48,243,48,253,39,253,27,241
,21,215,21,230,21,253,0
130 DATA 4 ,265,0,289,49,313,1,306,13,270,13
140 DATA 3 ,322,1,323,49,363,0,363,49
150 DATA 9 ,375,0,402,0,409,6,409,18,400,24,382,24
,375,30,375,43,381,49,409,49
160 DATA 2 ,420,0,420,49,465,49
170 DATA 1 ,420,25,451,25
180 DATA 3 ,527,49,479,49,479,1,526,0
190 DATA 1 ,479,24,516,24
200 DATA 8 ,541,1,541,48,572,48,582,39,582,29,572
,20,541,20,562,20,582,1
210 DATA 0 ,129,13
220 '
230 ' Read DATA
240 CLS: x=100: y=100: t=0: ee=0
250 '
260  READ n: s(t)=n: t=t+1
270  IF n=0 THEN 350' last data line
280  READ a,b: MOVE a+x,b+y
290  a(ee)=a: b(ee)=b: ee=ee+1
300  FOR i=1 TO n
310   READ a,b: DRAW a+x,b+y
320   a(ee)=a: b(ee)=b: ee=ee+1
330  NEXT: GOTO 260
340 '
350 READ a,b: MOVE a+x,b+y: FILL 2
360 a(ee)=a: b(ee)=b
370 '
380 ' press keys to manipulate
390 IF INKEY(2)=0 THEN r=0.9: GOSUB 620' shrink
400 IF INKEY(0)=0 THEN r=1.1: GOSUB 620' enlarge
410 IF INKEY(8)=0 THEN ag=10: GOSUB 680' rotate lt
420 IF INKEY(1)=0 THEN ag=-10:GOSUB 680' rotate rt
430 IF INKEY(39)=0 THEN r=0.9:GOSUB 580' compress
440 IF INKEY(31)=0 THEN r=1.1:GOSUB 580' stretch
450 IF INKEY(30)=0 THEN GOSUB 540' reflect in x
460 IF INKEY(22)=0 THEN GOSUB 500' reflect in y
470 IF INKEY(21)=32 THEN GOSUB 900' move
480 GOTO 390
490 '
500 ' reflect in y
510 FOR i=0 TO ee: b(i)=-b(i): NEXT
520 GOSUB 780: RETURN
530 '
540 ' reflect in x
550 FOR i=0 TO ee: a(i)=-a(i): NEXT
560 GOSUB 780: RETURN
570 '
580 ' compress/stretch
590 FOR i=0 TO ee: a(i)=a(i)*r: NEXT
600 GOSUB 780: RETURN
610 '
620 ' scale to shrink/enlarge
630 FOR i=0 TO ee
640  a(i)=a(i)*r: b(i)=b(i)*r
650 NEXT
660 GOSUB 780: RETURN
670 '
680 ' rotate by angle 'ag'
690 DEG
700 FOR i=0 TO ee
710  a=a(i): b=b(i)
720  q=ATN(b/(a+0.000001))
730  l=SQR(a*a+b*b): IF a<0 THEN l=-l
740  a(i)=COS(q+ag)*l: b(i)=SIN(q+ag)*l
750 NEXT
760 GOSUB 780: RETURN
770 '
780 ' redraw graphic
790 CLS: j=0
800 FOR i=0 TO t-1
810  n=s(i)
820  MOVE a(j)+x,b(j)+y: j=j+1
830  FOR k=1 TO n
840   DRAW a(j)+x,b(j)+y: j=j+1
850  NEXT
860 NEXT
870 MOVE a(j-1)+x,b(j-1)+y: FILL 2
880 RETURN
890 '
900 ' move around screen
910 IF INKEY(21)<>32 THEN RETURN ' no shift
920 IF INKEY(0)=32 THEN y=y+10: GOSUB 780' up
930 IF INKEY(2)=32 THEN y=y-10: GOSUB 780' dn
940 IF INKEY(8)=32 THEN x=x-10: GOSUB 780' lt
950 IF INKEY(1)=32 THEN x=x+10: GOSUB 780' rt
960 GOTO 910
```

Listing 1. *The 2D Transfer program.*

Graphics Techniques
- Part 5

Two-dimensional mazes are easy, so let's design a three-dimensional maze we can see ourselves move through!

WHAT COMES to mind when you think of a maze? Confusion? Tunnels and passages everywhere? Too hard to understand? In fact, we also associate mazes with intelligence. It is generally accepted that the greater the intelligence, the faster a path will be found through a maze. Scientists create many mazes for animals of various types to test their reactions and intelligence.

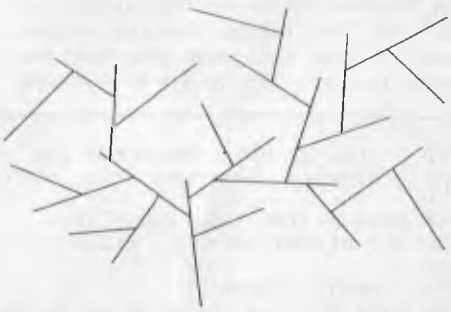


Figure 1. *The simplest of mazes – just like a tree, there's only one path between any two ends.*

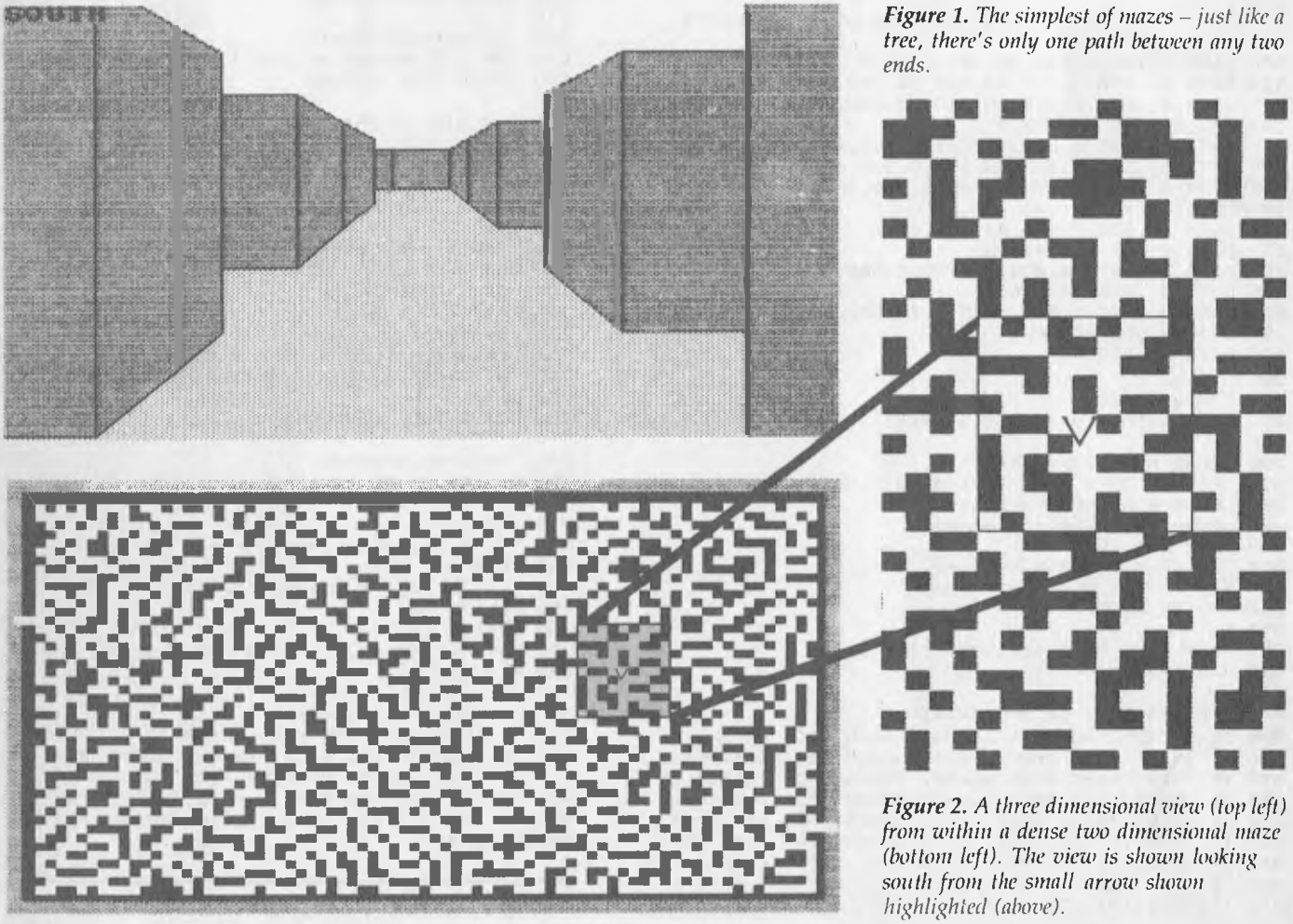


Figure 2. *A three dimensional view (top left) from within a dense two dimensional maze (bottom left). The view is shown looking south from the small arrow shown highlighted (above).*

For example, there have been many robot mice specifically designed and produced to find their way to a point in a maze, to advance the mechanical side as well as the intelligence side of robotics. Competitions are held every year or so to determine the fastest robot mouse to find its way through a maze. The fact is, mazes don't need to be confusing. When you understand what they are, and how to construct them, they become very interesting.

To start with, let's examine closely what a maze is. We know it is a complex network of paths or passages designed to puzzle those working through it. A similar system can be represented as a pattern of lines. They may range from round to square and include many shapes and sizes. One aspect common to almost all mazes is a connecting path which must be found between the beginning of the maze and the finish.

The perfect maze would be a compact, but easy to follow system of paths. It would have only one possible path from any point within the system to any other point, but with a path existing for every point. This type of maze may appear impossibly complex to be able to create in reality, but there is a simple technique which produces it.

Branching

First lets look at a system which already exists and has the above 'one path through' characteristic. For this we would just have to look at any tree. Notice that from any branch on the tree, to any other branch, there is really only one path. You simply move along the branch to the trunk and from there to the other branch. Look at a tree and try it for yourself, but make sure no one is watching you!

This type of branching can now easily be transformed into a simple procedure. To demonstrate this concept, simply draw a line on a piece of paper. From this line you then branch off in any direction. If you keep adding branches just anywhere and do not cross them over, you will have produced precisely the same kind of branching as on a tree. Notice that these branches are all the same width, the width of the pen, so that it is much harder to find your way from one branch to the next.

In replacement of pencil and paper, it is a simple matter to convert this procedure to computer – see Figure 1. Previously we have used straight lines, since the

```
100 ' MAZE DRAW. (c) 1988, Living Image
110 ' ____________________________________
120 '
130 'Setup screen 320 x 200 and array
140 MODE 1: DEFINT a-z
150 DIM j(2000),k(2000)' arrays for starting points
160 CLG 1 ' Clear graphics screen to ink 1, black.
170 '
180 'Draw outer edge in ink 3
190 FOR i=0 TO 10
200 PLOT i,i,3: DRAW 639-i,i
210 DRAW 639-i,399-i: DRAW i,399-i: DRAW i,i
220 NEXT i
230 '
240 'Overall starting point
250 sx=INT(RND*20): sy=201-sx*8: sx=sx*8+239: s=0
260 x=sx: y=sy
270 PLOT -10,-10, 0 ' Change plotting ink to 0, white
280 '
290 'Construct Maze
300 j(s)=x: k(s)=y: s=s+1
310 GOSUB 490' put square
320 le=le+1: IF le>20 THEN 430 'length >20 end branch
330 d1=0
340 a1=TEST(x+8,y-8): a2=TEST(x+8,y+8)
350 a3=TEST(x-8,y-8): a4=TEST(x-8,y+8)
360 ON 4*RND GOTO 380,390,400
370 d1=1: IF TEST(x+16,y)+a1+a2=3 THEN x=x+8: GOTO 300
380 IF TEST(x,y-16)+a3+a1=3 THEN y=y-8: GOTO 300
390 IF TEST(x-16,y)+a3+a4=3 THEN x=x-8: GOTO 300
400 IF TEST(x,y+16)+a4+a2=3 THEN y=y+8: GOTO 300
410 IF d1=0 THEN 370
420 'select new branching point from those stored
430 IF s=0 THEN 540
440 s=s-1: m=INT(RND*s)+1: le=0
450 x=j(m): y=k(m): j(m)=j(s): k(m)=k(s)
460 GOTO 330
470 '
480 'Wipe a small block out
490 PLOT x,y,: DRAW x+6,y: PLOT x+6,y-2: DRAW x,y-2
500 PLOT x,y-4: DRAW x+6,y-4: PLOT x+6,y-6: DRAW x,y-6
510 RETURN
520 '
530 'Put Finish and Start
540 x=619: y=INT(RND*10)*8+22
550 IF TEST(x-8,y)<>0 THEN 540
560 y=y+2: x=x+4: GOSUB 490: x=x+8
570 GOSUB 490
580 x=27: y=INT(RND*10)*8+206
590 IF TEST(x+8,y)<>0 THEN 580
600 y=y-6: x=x-12: GOSUB 490: x=x-8
610 GOSUB 490
```

Listing 1. *Maze Draw is a simple but efficient routine for drawing a two dimensional maze – it uses the Test command to test points on the screen and a branching point array of 2000 points.*

branches on a tree are usually fairly straight. However, if we use crooked lines instead, our lines will form more of a maze. It then becomes an arduous task to find the correct passage through. Of course, the lines still must not cross or touch other branches. Each branch still touches another only once.

On a computer, a maze is usually constructed by first drawing a grid of small boxes over the screen. An array is set up to store the state of every edge on the grid. Now as the edges are taken away, a passage is constructed along the boxes. Branching is made by forming new passages off the old ones. The computer can check if a passage exists by checking the array which is changed as the passages are constructed.

Amazing improvements

A major consideration in the construction of mazes is where the beginning of the first branching is going to occur. This point would be picked randomly, but far away from the start and end points. The beginning and end should be as far from each other as possible to increase the length and difficulty level of the path.

Notice, too, that while a very large maze takes much longer to construct than a small one, it will be much more difficult to traverse. The size of the maze is normally limited by the screen resolution and memory capacity, as well as the speed of the computer.

Instead of using an array, the computer can simply check the state of the screen using a screen test command. This eliminates the need for arrays. Using the screen to test the maze also means that very large, complex mazes can be constructed on a high resolution graphics screen and little memory is being used. Array storage can be very heavy on memory when large, complex mazes are constructed.

Another big improvement is to do away with the grid system all together and simply construct the maze 'freestyle'. This enables a more complex maze to be made in a smaller space. A different checking system though, is obviously needed to scan the screen for other passages already constructed. However, with greater complexity comes a problem: the computer will have to work harder to find suitable branching places. No need to worry though, this is overcome with the next, most important improvement.

Many maze generating programs simply sample random spots in the maze to find a branching point which is suitable. This causes huge delays, especially when the maze has nearly filled its allotted space and there are few branching points left. There have been many methods to overcome this and the following one seems as good as any other.

As the paths are constructed, block by block, the computer stores every point of every path. Then when the current branch stops, the programs simply picks a branching point out of the array and either starts from there, if it is legitimate, or eliminates the point out of the array and reduces the number of stored points. When the array runs out, then there are no more possible branches and the computer has completely filled the space.

It seems that with this branch/storage method an extremely large array of points will be needed, but in reality an array of about 2000 points is usually enough. When the 2000th point is stored, then the current branch stops and many of the old, useless positions are thrown out before another branch begins.

The program given in Listing 1, Maze Draw, uses all of the improvements described so far. It uses the Test command to test points on the screen and a branching point array of 2000 points. A larger array may enable the program to proceed faster. Carefully type the program into your computer and observe how the array

```
1000 '---Turn Maze into 3D display-----------------
1010 '
1020 ' Load maze into array
1030 DIM x(15),t(15),b(15), m(80,50)
1040 FOR i=0 TO 80
1050  FOR j=0 TO 50
1060   m(i,j)=1
1070   IF TEST(i*8,j*8)=0 THEN m(i,j)=0
1080  NEXT j
1090 NEXT i
1100 '
1110 ' Calculate perspective points and store
1120 dx=250: tx=280: bx=400: x=-dx/3
1130 x(0)=x: t(0)=250+tx: b(0)=250-bx
1140 s!=1.62
1150 FOR i=1 TO 15
1160 dx=dx/s!: x=x+dx: x(i)=x
1170 tx=tx/s!: t=250+tx: t(i)=t
1180 bx=bx/s!: b=250-bx: b(i)=b
1190 NEXT
1200 '
1210 ' Setup display
1220 MODE 1: INK 0,13: INK 1,0
1230 BORDER 9: INK 2,5: INK 3,20
1240 x=RND*50+10.5: y=RND*30+10.5
1250 IF m(x,y)<>0 THEN 1240
1260 PLOT -10,10,1
1270 dx=0: dy=-1
1280 '
1290 ' Main loop; display & wait for key presses
1300 GOSUB 1470 ' redraw 3d
1310 LOCATE 1,1
1320 IF dy=-1 THEN PRINT "SOUTH";
1330 IF dy=1 THEN PRINT "NORTH";
1340 IF dx=-1 THEN PRINT "EAST";
1350 IF dx=1 THEN PRINT "WEST";
1360 IF INKEY(0)<>0 THEN 1390
1370 IF m(x+dx,y+dy)=0 THEN x=x+dx: y=y+dy: GOTO 1300
1380 IF m(x+dx,y+dy)=1 THEN SOUND 1,300
1390 IF INKEY(8)=0 THEN tt=dx: dx=dy: dy=-tt: GOTO 1300
1400 IF INKEY(1)=0 THEN tt=dx: dx=-dy: dy=tt: GOTO 1300
1410 IF INKEY(2)<>0 THEN 1440
1420 IF m(x-dx,y-dy)=0 THEN x=x-dx: y=y-dy: GOTO 1300
1430 IF m(x-dx,y-dy)=1 THEN SOUND 1,300
```

```
1440 GOTO 1360
1450 END
1460 '
1470 ' 3d Redraw Subroutene
1480 CLS
1490 dist=1
1500 xx=x+dx: yy=y+dy
1510 WHILE m(xx,yy)=0
1520 dist=dist+1: xx=xx+dx: yy=yy+dy
1530 WEND
1540 IF dist>15 THEN dist=15
1550 ' vertical lines
1560 FOR i=1 TO dist
1570 MOVE x(i),t(i): DRAW x(i),b(i)
1580 MOVE 640-x(i),b(i): DRAW 640-x(i),t(i)
1590 NEXT
1600 ' horizontal at end
1610 MOVE x(dist),t(dist): DRAW 640-x(dist),t(dist)
1620 MOVE x(dist),b(dist): DRAW 640-x(dist),b(dist)
1630 '
1640 FOR i=0 TO dist-1
1650 '
1660 'find block on left side
1670 IF dy=-1 THEN m=m(x-1,y-i)
1680 IF dy=1 THEN m=m(x+1,y+i)
1690 IF dx=-1 THEN m=m(x-i,y+1)
1700 IF dx=1 THEN m=m(x+i,y-1)
1710 '
1720 'draw horizontal if a passage exists
1730 IF m<>1 THEN 1770
1740 MOVE x(i),b(i): DRAW x(i+1),b(i+1)
1750 MOVE x(i),t(i): DRAW x(i+1),t(i+1)
1760 GOTO 1810
1770 MOVE x(i),b(i+1): DRAW x(i+1),b(i+1)
1780 MOVE x(i),t(i+1): DRAW x(i+1),t(i+1)
1790 '
1800 ' find block on right side
1810 IF dy=-1 THEN m=m(x+1,y-i)
1820 IF dy=1 THEN m=m(x-1,y+i)
1830 IF dx=-1 THEN m=m(x-i,y-1)
1840 IF dx=1 THEN m=m(x+i,y+1)
1850 '
1860 ' draw horizontal line if passage exists
1870 IF m<>1 THEN 1910
1880 MOVE 640-x(i),b(i): DRAW 640-x(i+1),b(i+1)
1890 MOVE 640-x(i),t(i): DRAW 640-x(i+1),t(i+1)
1900 GOTO 1930
1910 MOVE 640-x(i),b(i+1): DRAW 640-x(i+1),b(i+1)
1920 MOVE 640-x(i),t(i+1): DRAW 640-x(i+1),t(i+1)
1930 NEXT i
1940 '
1950 ' fill floor and sky with colour
1960 MOVE 320,0: FILL 2
1970 IF dist>1 THEN MOVE 320,399: FILL 3
1980 RETURN
```

Listing 2. This is an extension of Listing 1 that uses the newly constructed maze to position you randomly within it and then constructs a three dimensional view from that position. The can be changed by stepping forward or backward or by turning to the left or right.

is constructed. A delay between lines 310 and 320 may be inserted so that you can see the process more clearly.

Three dimensions

Once a maze has been created, it can be used in various ways. A simple two dimensional representation on the screen is the most common. Sometimes the screen only displays a small section of the maze with most of the maze being off the screen. This is more difficult to program, but it is possible to have a much larger overall maze. Another way to extend the perception of the maze is to give a three dimensional scene of what it would look like if we were actually in the maze itself, looking down a corridor for instance, with side passages branching off.

A maze or any passage or array, can be transferred into a three dimensional view by projecting the picture as if you were standing inside the passage, room or maze. Side passages are easy to build on just by making lines horizontal instead of diagonal. Now, look at Figure 2, which shows both a maze in two dimensional form, and a three dimensional view from within the same maze.

In the view, all vertical lines stay the same, in precalculated perspective, until the other end of the passage is reached. The side walls follow this line of perspective unless a side passage exists, in which case a horizontal line is drawn as the corner of the passage, both above and below.

Listing 2 is an extension of Listing 1. It will take the newly constructed maze and position you randomly within it. Then a view is constructed of the inside of the maze in three dimensions. By pressing four keys you may step either forward, backward or turn to the left or right. The display will then quickly redraw the new view. The directions, North, South, East and West are given with the top of the maze taken as being North.

Probably many uses for mazes and particularly the three dimensional passage section have already come to mind. Programs such as adventure games are virtually half done with the above system. The 'passages' program may even be expanded to duplicate the movements through a building design to find.

In conclusion, note the use of random numbers in the maze drawing process. Computer graphics are relying more and more on random and semi-random constructions. Random numbers are going to play an increasing role in computing as programs become more complex. In fact, these random functions are extending into every avenue of computing. □

THE LES BELL

NCYCLOPEDIA OF

ATCH

ILE

ROGRAMMING

We all thought Les was a dedicated CP/M hacker – not so, he insists; he just spent more time using CP/M and more time programming for DOS. To prove his point, he's put together a batch of techniques he's been using to make life easier . . .

BATCH FILES? What are they? Batch files are text files containing, at the elementary level, sequences of DOS commands and programs to be run. The most common use of batch files is to ensure that a sequence of commands is executed correctly without relying on the user's memory to get it right.

For example, some software packages require you to use the back-up command on the subdirectories only, and not back up the root directory of their hard disk. In this case, you could perform the job manually by issuing a separate back-up command for each subdirectory:

```
C:\FW>BACKUP C:\BIN A: /S
C:\FW>BACKUP C:\FW A: /S
C:\FW>BACKUP C:\DB3 A: /S
C:\FW>BACKUP C:\C86 A: /S
```

This is fine as long as you have a good memory and can be sure you (or the user for whom you have set up the PC) will remember to back up all the subdirectories. You might also wonder if you have the patience to type in all those commands — but you'll have to sit there anyway, feeding floppy disks into the machine, so typing the commands is not a great inconvenience. However, if the back-up commands worked without human intervention, waiting around just to type the commands would be a real pain in the ...

You can solve the problem with a batch file; just create a text file called BACKALL.BAT, with the back-up commands in it:

```
BACKUP C:\BIN A: /S
BACKUP C:\FW A: /S
BACKUP C:\DB3 A: /S
BACKUP C:\C86 A: /S
```

You can do this with your favourite editor or word processor (provided it can edit ASCII text) or with EDLIN if you have to, or you can simply copy the file from the keyboard into the file:

```
C:\FW>COPY CON BACKALL.BAT
BACKUP C:\BIN A: /S
BACKUP C:\FW A: /S
BACKUP C:\DB3 A: /S
BACKUP C:\C86 A: /S
^Z
<- press Ctrl and Z together here
1 File(s) Copied
C:\FW>
```

Now, typing the name of the batch file will invoke it, causing DOS to execute each of the back-up commands in turn –

```
C:\FW>BACKALL

C:\FW>BACKUP C:\BIN A: /S
     (etc)
      .
      .
      .
C:\FW>
```

There are a few rules about the names you give batch files or, more particularly, about the way DOS searches for program files (which includes batch files).

When you give a command at the DOS prompt, like this:

C: FW>DO SUMMAT

you are presumably trying to invoke a program called DO, to operate on the file SUMMAT. DOS will search the current (sub)directory for the file DO.COM and, if it finds it, will load and run the program. If DO.COM is not found, then DOS searches again for DO.EXE and will load and run this program if possible.

If neither of these files is found, DOS searches for a batch file called DO.BAT, and if this is found, it invokes the batch file processor and runs DO.BAT.

If none of these files is found, DOS will then repeat this search pattern for each of the (sub)directories named in the current search path. This can be displayed and set using the PATH command.

To summarise, DOS's search process is as follows –

1. In the current directory, search for a .COM file;
2. In the current directory, search for a .EXE file;
3. In the current directory, search for a .BAT file;
4. For each subdirectory in the PATH –
   a. Search for a .COM file,
   b. Search for a .EXE file,
   c. Search for a .BAT file;
5. If none of the above succeeds, print an error message.

The implication for batch file naming is that you cannot give a batch file the same name as an existing .COM or .EXE file, unless that other file comes later in the search path. For example, we could not have named our back-up batch file BACK-

UP.BAT, since then the BACKUP commands in the batch file would simply reinvoke the batch file repeatedly, causing the system to hang up.

Under DOS 3.0 and later, it is possible to give batch files and programs the same name and distinguish between them by prefixing the name of each with the appropriate path, or by using the SUBST command. I'll give an example of this later.

Elementary Batch Commands

By simply placing a list of DOS commands or programs to be invoked into a batch file, you can automate the most common procedures under DOS. However, the results are not terribly elegant, nor can you do things like prompting the user to change disks. However, DOS provides a number of commands, in most cases special to the batch processor and not available from the DOS prompt, which make these things possible. The first three of these are ECHO, REM and PAUSE.

ECHO has two purposes. First, it can be used to turn on and off the echoing of DOS commands to the console. In the example above, each separate BACKUP command was visible, making it clear that a series of programs was being run, rather than a single one. To make it look like a single program, add ECHO OFF at the beginning of the line. This suppresses the echoing of commands to the screen. Naturally, ECHO ON turns it on again, and when the batch file ends, echoing is turned on again for the next batch file.

The second use of ECHO is to display messages for the user. For example,

ECHO Please insert transfer disk in A:

will display the message

'Please insert ... A:'

on the screen.

There is a bug in the batch file processor: if the last line of your batch file does not have a carriage return at the end, it will be echoed anyway, regardless of the state of the ECHO switch.

The REM command is similar to ECHO, except it is used to insert comments into the batch file. If ECHO is set ON, the REM comments will appear as the batch file runs, but if ECHO is OFF, the REM comments do not appear.

Finally, the PAUSE command displays the message

Strike a key when ready ...

and halts execution temporarily. This is used to allow you to change disks, and is usually preceded by some kind of ECHO message.

What's my Percentage? — Elementary Parameters

A common use for batch files is to run compilers. For example, I commonly use the Computer Innovations Optimising C86 C compiler, which is actually four separate programs which have to be run in sequence. So, to compile the program WX2.C, I must give the commands:

```
C:\C86>CC1 WX2
C:\C86>CC2 WX2
C:\C86>CC3 WX2
C:\C86>CC4 WX2
```

(the output of the compiler passes is not shown.) Obviously, this is an ideal candidate for a batch file, called MAKEWX2.BAT:

```
ECHO OFF
CC1 WX2
CC2 WX2
CC3 WX2
CC4 WX2
```

There are a couple of problems with this batch file. First, what if I now want to compile the program SR.C? Do I now have to create a new batch file with different commands called MAKESR.BAT which would look like –

```
ECHO OFF
CC1 SR
CC2 SR
CC3 SR
CC4 SR
```

Furthermore, what if the first pass of the compiler discovers a syntax error in my program? It will produce an error message, telling me what's wrong, but then CC2, 3 and 4 will run, and they will either produce error messages, pushing the first, helpful, message off the screen or (much worse) they will run normally on the files left over from the previous compilation, pushing the error message off the screen and leaving me with an unchanged program!

Let's deal with the first problem. Batch files can be created which do not have the names of files and disk drives embedded within them, but which instead are given these parameters at the time they are run. A parameter is something that changes each time the batch file is run, and which is given to the batch file by typing it on the command line which runs the batch file.

To do this, we modify either of the MAKE???.BAT files above, by replacing the filenames with a parameter. The resulting batch file, MAKE.BAT, looks like this:

```
ECHO OFF
CC1 %1
CC2 %1
CC3 %1
CC4 %1
```

The '%1' symbols stand for the first word on the command line after the batch file name itself (this can be referred to as %0). So, to run this batch file to compile WX2.C, we invoke it with the command line

C: C86>MAKE WX2

while to use it to compile SR.C, we type

C: C86>MAKE SR

In each case, the word after 'MAKE' on the command line (WX2 or SR) replaces %1 in the batch file, which then issues the correct commands to compile the appropriate file. You can see that this is so by removing the ECHO OFF command.

Advanced Batch Commands

You can make batch files even more powerful by using some batch commands which are more usually found in programming languages. These allow you to test for various conditions or to loop around, repeatedly executing commands on different files.

The first command is the GOTO command. This transfers control to the command after the label which is the target for the GOTO. A label consists of a colon (:) followed by a word, as the first thing on the line of the file. Here's a short (nonsensical) example:

```
ECHO OFF
ECHO This should be printed
GOTO OVER
ECHO This should never be printed
:OVER
ECHO Now we're finished
```

Obviously, GOTO by itself is not much use; it has to be used in conjunction with other statements. In fact, in batch files, the only command which is used in conjunction with GOTO is the IF command, which comes in several variations.

The first IF variation allows you to test whether a file exists; this is useful in backup batch files, compiler control batch files and others. Let's see how it can be used with Lattice C, for example, to take care of errors in compilations:

```
ECHO OFF
ERASE %1.Q
ERASE %1.OBJ
LC1 %1
IF NOT EXISTS %1.Q GOTO ERR
LC2 %1
IF NOT EXISTS %1.OBJ GOTO ERR
LINK C+%1,%1,,LC.LIB
GOTO OVER
:ERR
ECHO Error compiling %1.C
:OVER
```

Here there are two passes of the compiler; the first one reads a file with a .C filetype and produces a corresponding .Q file, while the second reads the .Q file and produces a .OBJ file, which is then linked with the C compiler library to produce the desired .EXE file.

If either pass of the compiler discovers an error, it will not produce its output file, and we rely on this fact to control the batch file and jump to a command which prints an error message. Notice, if everything works okay, we have to jump past the error message so it doesn't appear.

Notice also that although the compiler may discover an error and not produce an output file, there may be an existing output file from a previous compile. We must therefore delete any such files before starting, otherwise our logic won't work.

Finally, notice that the IF EXISTS test can be modified to work 'backwards' by using the 'NOT' modifier, which makes it perform a GOTO if the specified file does not exist. The NOT modifier can also be applied to the other IF tests.

There is a better way of testing for errors after programs have run, but the programs have to be written specially to take advantage of it. This uses a special 'system variable' called the ERRORLEVEL, which is set by some programs as they return control to the operating system: a value of zero (0) indicates normal program termination with no errors, while a value of one (1) or higher indicates some error. In general, the higher the ERRORLEVEL value, the more severe the error.

Batch files can test for errors with the IF ERRORLEVEL statement. For example:

IF ERRORLEVEL 1 GOTO ERR

but notice that this is read as 'if errorlevel greater than or equal to one goto err'. In other words, execution will continue normally if the ERRORLEVEL is zero, but will branch to :ERR if it is one or greater. Here's an example of a smarter batch file for the C86 compiler:

The implication for batch file naming is that you cannot give a batch file the same name as an existing .COM or .EXE file, unless that other file comes later in the search path.

```
ECHO OFF
CC1 %1
IF ERRORLEVEL 1 GOTO ERR
CC2 %1
IF ERRORLEVEL 1 GOTO ERR
CC3 %1
IF ERRORLEVEL 1 GOTO ERR
CC4 %1
IF ERRORLEVEL 1 GOTO ERR
LINK %1,%1,,C86S2S
GOTO OVER
:ERR
ECHO Error compiling %1
:OVER
```

This works because each pass of the Computer Innovations compiler sets the ERRORLEVEL as it exits.

The final variation on the IF command allows comparison of strings. Its basic format is:

IF 'string1'=='string2'

There are a couple of points to notice about this. First, the two strings must be enclosed in quotes, and secondly the comparison operator is '==', that is, double equals signs and not a single equals like in most other languages.

This can be used in batch files in a variety of ways. For example, a batch file which cleans up a subdirectory by copying files to a floppy disk might optionally delete .BAK files like this:

IF '%3'=='DELETE' ERASE *.BAK

Notice the parameter must be enclosed in quotes, since it is simply picked up off the command line when the batch file is invoked, like this:

C: C86>CLEANUP *.DOC A: DELETE

Another use for string comparison is testing to see whether the user has supplied certain command-line parameters. For example, Figure 1 shows the full batch file, CC.BAT, which I actually use to drive the Computer Innovations compiler. Notice the use of the test –

if '%1'==" goto explain

which tests to see if %1 is a null string, and if so, explains the correct usage of the batch file.

The final complex batch command is FOR. This is similar in some respects to the FOR loop in BASIC, in that it allows repetitive looping inside batch files — and more. Its basic format is –

FOR pvarname IN (parmlist) DO command

where pvarname is a pseudo-variable name and parmlist is a list of parameters, such as filenames. A pseudo-variable name is rather like a command-line parameter, except that it is not just numbered, it is also named, and takes the form of two per cent signs and the name: %%-name. The parameter list can include filenames, keywords, commands, and even command-line parameters like %1, %2 and so on.

For example, suppose we wanted to separately compile and then link together three C language source files. We could do it by removing the link command from the batch files shown above, running the batch file three times and then giving the link command manually.

A better way to do it, at the expense of

```
echo off
if '%1'=='' goto explain
cc1 %1 %2 %3 %4 %5
if errorlevel 1 goto err
cc2 %1
if errorlevel 1 goto err
cc3 %1
if errorlevel 1 goto err
cc4 %1
if errorlevel 1 goto err
link %1,,NUL,c86s2s
goto done
:err
echo error compiling %1
goto done
:explain
eho usage:-
echo    cc file [flags]
echo    example:- cc hello.c -hc:\c86\,c:\ -i -x
echo    this batch file assumes that you are using
echo    the c86s2s library (small model, dos2, software f.p.)
echo    and LINK.EXE to link
:done
```

Figure 1

error checking, would be to construct a batch file using the FOR command as shown in Figure 2. This will compile the three files PROG1.C, PROG2.C and PROG3.C, and then link them with the compiler's library to produce the program MYPROG.EXE. Of course, a more versatile way to do this would be to create the batch file shown in Figure 3, which I'll call MAKE3.BAT, which allows you to invoke the file with a command line like

C: C86>MAKE3 IMAIN SFUNCS IFUNCS INDEX –2 –F

This will compile the first three files (IMAIN.C, SFUNCS.C and IFUNCS.C) to produce the program INDEX.EXE. The –2 and –F parameters are command line options for the compiler, causing it to produce code optimised for the 80286 processor and use 'frugal' optimisation, respectively.

The price paid for this flexibility is error checking, but we shall investigate ways to improve this later.

The SHIFT command is the last command supported by the DOS batch processor. With standard batch file parameters you can only have 9 parameters, plus the batch filename; that is, %0 to %9. %10 is viewed as %1 immediately followed by a 0. The SHIFT command has the effect of discarding the current %0 (you probably didn't need it anyway) and putting %1 in its place. Then %2 is shifted into %1, %3 into %2, and so on until finally what would have been the inaccessible %10 is shifted into %9.

Now, batch files with 10 or more parameters are a bit mind-boggling to contemplate, but SHIFT could be used in conjunction with FOR loops. However, I've never needed it so far.

```
ECHO OFF
FOR %%VAR IN (PROG1 PROG2 PROG3) DO CC1 %%VAR
FOR %%VAR IN (PROG1 PROG2 PROG3) DO CC2 %%VAR
FOR %%VAR IN (PROG1 PROG2 PROG3) DO CC3 %%VAR
FOR %%VAR IN (PROG1 PROG2 PROG3) DO CC4 %%VAR
LINK PROG1+PROG2+PROG3,MYPROG,,C86S2S
```

Figure 2

```
ECHO OFF
FOR %%VAR IN (%1 %2 %3) DO CC1 %%VAR %5 %6 %7
FOR %%VAR IN (%1 %2 %3) DO CC2 %%VAR
FOR %%VAR IN (%1 %2 %3) DO CC3 %%VAR
FOR %%VAR IN (%1 %2 %3) DO CC4 %%VAR
LINK %1+%2+%3,%4,,C86S2S
```

Figure 3

```
echo off
:start
cls
echo                    Main Menu
echo  1 - Run WordStar
echo  2 - Compile choose.c    <
choose
if errorlevel 2 goto mc
if errorlevel 1 goto dows
if errorlevel 0 goto done
:mc
command /c cc choose
goto start
:dows
command /c ws
goto start
:done
```

Figure 4.

```
/* CHOOSE.C - set ERRORLEVEL interactively */
/* Programmed by LB */

#include "stdio.h"

main()
{
        int choice;
        static char temp[] = {" "};

        if((stdin = fopen("CON","rb")) == NULL) {
                abort("ask: Unable to open console stream");
        }

        fputs("Please enter your choice: ",stdout);
        while(!isdigit(temp[0] = getchar()));
        choice = atoi(temp);
        putchar('\n');
        exit(choice);
}
```

Listing 2

```
/* ASK.C - set ERRORLEVEL interactively */
/* Programmed by LB */

#include "stdio.h"

main()
{
        int answer;

        if((stdin = fopen("CON",'rb")) == NULL) {
                abort("ask: Unable to open console stream");
        }

        fputs("Press Y or N: ",stdout);
        while(((answer = toupper(getchar())) != 'Y') && (answer != 'N'));

        putchar('\n');
        if (answer == 'Y') exit(0);
        else if (answer == 'N') exit(1);
}
```

Listing 1

```
REM Loop processor for various files
ECHO OFF
FOR %%X IN (CH1.DOC CH2.DOC CH3.DOC CH4.DOC CH5.DOC) DO CD %%X
REM CD.BAT - Copy file then delete it
COPY %1 \ARCHIVE\%1
DEL %1
```

Figure 5.

Extending Batch Processing

The IF and FOR commands give considerable flexibility in controlling the execution of individual commands. To control the execution of sequences of multiple commands, we have to use GOTO, which the experienced programmers amongst us recognise as **A Bad Thing**.

The lynchpin of good programming practice is the ability (and the desire) to have subroutines or procedures. This applies to all programming languages, from BASIC to C and PL/I, including DOS batch files.

Now, surely in batch files all you have to do to get subroutines is have one batch file call another? Sounds fine in theory. We could write a pair of batch files to, for example, copy files to a new subdirectory and then delete the old version as shown in Figure 5.

If you try this, you'll find it doesn't work. Naming one batch file inside another works like a GOTO; control transfers to the named batch file but then never returns, so the batch file terminates at the end of the 'subroutine' batch file.

How, then, can you call one batch file from within another? You can, but the technique is well hidden in the DOS manuals. The secret is that each batch file requires its own copy of the batch file processor to run correctly. When it terminates, its copy of the batch file processor disappears from memory and control returns to the batch processor for the calling batch file, which then resumes where it left off.

The batch file processor is part of the resident portion of the COMMAND.COM program. You can invoke a new copy of COMMAND.COM and pass it a command (like a batch file name) by using the command

COMMAND /C command line

A version of the batch files given above which would work is shown in Figure 5. Notice there is an ECHO OFF statement in both batch files; this is because CD.BAT starts running under a completely fresh copy of COMMAND.COM which has ECHO set ON.

In practice, copies of COMMAND.COM inherit a lot of information about the current DOS set-up — it's just that the current state of ECHO is not part of that. In particular, COMMAND.COM manages an area of memory called the environment, in which various strings are stored which control the way DOS operates. You can see these strings by typing SET and pressing return; this will display the contents of the environment:

```
C:\C86>SET
COMSPEC=C:\COMMAND.COM
PATH=C:\;C:\BIN;C:\C86
PROMPT=$p$g
C86TEMP=f:
```

As you can see, all environment strings take the form varname=string. Some are obvious, like PROMPT=, which saves the current prompt string which sets the DOS prompt, and PATH=, which is set by the PATH command in your AUTOEXEC.BAT file (what do you mean, you don't have a PATH command in your AUTOEXEC.BAT file?). Others are specific to particular packages, like the C86TEMP variable, which specifies where the C86 compiler will put its temporary files (drive F: is a memory disk, for speed). Finally, the COMSPEC string, which is present on all systems, specifies from where COMMAND.COM will reload its non-resident portion if it should be overwritten.

The important point is that copies of COMMAND.COM receive a copy of their parent's environment, and therefore have access to all these variables. But because it is a copy of the environment, changing any variables will have no effect on the parent's environment. In other words, variables are local, not global.

However, there is one way in which a copy of COMMAND.COM can affect its parent, and that is through the ERRORLEVEL variable. When a copy of COMMAND.COM terminates, it returns its current ERRORLEVEL value to its parent, so that if this batch file encountered an error the calling batch file knows about it, too.

It is possible to make batch files interactive, in other words, to make them ask questions. The easiest way to do this is with the ERRORLEVEL, by writing a short program which asks for a yes or no response and sets the ERRORLEVEL accordingly. Such a program, ASK.COM, is in the public domain, in the PC/Blue user group public domain library and available from user groups around the country.

ASK.COM asks the user for a Y/N response and sets the ERRORLEVEL to 0 for a Y response and 1 for N. This can then be tested by an IF command. For example, I use this technique in my AUTOEXEC.BAT file:

```
echo off
path c:\;c:\bin;c:\c86
xtime
prompt $p$g
set c86temp=f:
subst g: c:\bin
echo Want Sidekick loaded?
ask
if errorlevel 1 goto nosk
sk
:nosk
```

This sets things up on my system, and then asks if I want Sidekick loaded into memory. If I answer with a Y, then SK.COM is run, otherwise execution jumps past this point. This technique can be extended to allow menus to be constructed.

Listing 1 shows a C version of the ASK program, written for Computer Innovations Optimising C86. This version compiles to rather a large size at 10 Kbytes — not really a problem, but the public domain version is rather neater at 256 bytes. The problem is that I'm too lazy to write any assembly language if I can avoid it: the C version took 10 minutes to write, while an assembler version would have taken me hours.

Likewise, Listing 2 shows a C program called CHOOSE.C. This allows you to enter a single digit as a menu choice, and then sets the ERRORLEVEL to that value. This program could easily be extended to allow selections higher than 9. Figure 4 shows a menu batch file which uses the CHOOSE.EXE program. □

THE LES BELL ENCYCLOPEDIA OF BATCH FILE PROGRAMMING PART 2

In Part 1, Les showed how to make Batch Files interactive. But, a more powerful trick is to use environmental variables.

YOU LEFT ME in Part I engrossed in making batch files interactive, using the public domain program ASK.COM. More powerful is the ability to use environment variables within batch files. There are several ways this can be done.

The simplest way to set an environment variable is to set it using the SET command. For example:

```
SET USER=Fred
```

In typing a SET command, do not type any spaces around the equals sign — they seem to upset the command. The command is not case sensitive; the variable name is always transferred into upper case when it is stored, while the value is stored exactly as you type it.

You can type a SET command before running the batch file, or alternatively you can incorporate the SET command into the batch file. However, there is a restriction on SETting environment variables from within batch files: the operation of the batch file processor restricts the growth of the environment, so that only a few strings can be stored there. By contrast, if you keep setting strings into the environment manually, it can expand up to 32 Kbytes in size.

There is a public domain utility (I forget what it's called, unfortunately) which does for environment variables what ASK does for the ERRORLEVEL. In other words, it is like a BASIC INPUT statement for batch files. By the time this appears in print, you can expect to see either that program or one of mine that does the same thing on the *Your Computer* Bulletin Board. I would have had one written in C except that my C compiler can only read environment

variables, not write them.

Having got a value stored in an environment variable, how do you access it from within a batch file? The answer (not described anywhere in the DOS documentation, incidentally) is yet another variant on the use of percentage signs. To extract the value of variable 'envvar' in a batch file, simply refer to it as '%envvar%.' For example, in a user logon batch file:

IF '%HELP%'=='ON' TYPE LOGON.HLP

This will type a help file should the user require it. You can similarly store passwords and other information in the environment.

Wildcard Expansion

Several DOS commands, and other programs, do not accept wildcards in the filenames passed to them. For example, the DOS FIND command, a very useful little Chinese copy of the UNIX grep command, accepts a list of filenames, but no wildcards, which is rather inconvenient. Its syntax is:

FIND [flags] "string" filelist

This means that to search all .LET files for a particular name, you would have to give the full list of all filenames — in other words, run a DIR command to find all the .LET files first. Isn't this the kind of repetitive task the computer is supposed to do for you?

A solution comes to us courtesy of the DOS FOR batch command. Wildcards are permissible in the parameter list of the FOR command, and are automatically expanded. So, we can easily create a batch file to search all .LET (or any other type) files for a string:

```
ECHO OFF
FOR %%V IN (%2) DO FIND %1 %%V
```

If we call this batch file GREP.BAT (cheeky, but meaningful!) we can invoke it with a command like

```
C: C86>GREP "printf" *.C
```

which will search all the C source files in this subdirectory to find those containing calls to the printf function. Likewise:

```
C: C86>GREP "Smith" *.LET
```

will find all letters which mention a person called Smith.

Redirection and Batch Files

If you want to save all the output from the GREP batch file into a text file for later perusal with a word processor or editor, you cannot simply type –

```
C: C86>GREP "Smith" *.LET >GREP.OUT
```

This will just create an empty GREP.OUT file, and the output of the FIND commands will go to the screen as usual. Instead, the redirection must be applied inside the batch file. However, you cannot simply use output redirection, because each time the FIND command is run, its output will overwrite the file created by the previous run — so only the output from the last FIND will be saved.

To get around this, we must use **append redirection**, which sends its output to the end of the target file, after any existing material. Now the problem is that if we write append redirection into the batch file, it will simply append its output onto any previous run of the batch file, instead of creating a fresh, new file. No problem: I had already written a program called MAKE.C which creates an empty file, in connection with a command logging system I had developed — see Listing 1.

```
/* Make empty file for history and similar utilities */

#include "stdio.h"

main(argc,argv)
int argc;
char **argv;
{
        FILE *f;

        if(argc != 2) {
                printf("Usage: make filename");
                exit(1);
        }
        if((f = fopen(argv[1],"w")) == 0) {
                printf("make: Unable to make file %s",argv[1]);
                exit(1);
        }
        fclose(f);
}
```

Listing 1. MAKE.C

Listing 2 shows a rather smarter batch file, which will send its output to the screen or to an optional output file. It is left as an exercise to the reader (shades of school textbooks!) to add optional append or overwrite of the file %3.

```
echo off
if '%3'=='' GOTO nosave
make %3
for %%v in (%2) do find %1 %%v >> %3
goto end
:nosave
for %%v in (%2) do find %1 %%v
:end
```

Listing 2.

Redirection is useful for other purposes, too. For example, in a batch file, you may want to change the PATH searched for program files, and then set it back the way it was later. How can you store the current PATH setting?

The answer is this technique:

```
PATH >OLDPATH.BAT
.
(batch processing)
.
COMMAND /C OLDPATH
```

The batch file OLDPATH.BAT will contain a single line of the format

```
PATH=C: ;C: BIN;C: C86
```

or whatever applies to your system. When it is executed it sets the path back the way it was.

Dynamic System Configuration

A common problem with departmental PCs is the sharing of PCs between users who have conflicting requirements. For example, one user might use the machine with dBase III, and might want a 256 Kbyte memory disk drive, while the other user wants as much main memory as possible in order to run large Lotus 1-2-3 spreadsheets.

Since (under DOS 3.1) the memory drive is set up by the CONFIG.SYS file, changing the system around means editing that file, saving it and then rebooting. This process is tedious and prone to error but it can be automated thus –

First, create the two different CONFIG.SYS files, in your BIN subdirectory, under two different names. For example, LOTUS.SYS:

```
COUNTRY=061
```

and DBASE.SYS:

```
DEVICE=VDISK.SYS 256
BUFFERS=30
FILES=20
COUNTRY=061
```

Now, create two batch files, as follows. First, DBASEMEM.BAT:

```
ECHO OFF
COPY BIN DBASE.SYS CONFIG.SYS
REBOOT
```

and then LOTUSMEM.BAT:

```
ECHO OFF
COPY BIN LOTUS.SYS CONFIG.SYS
REBOOT
```

giving the command

```
C: BIN>LOTUSMEM
```

will copy the appropriate contents into CONFIG.SYS and then reboot the system, while

```
C: BIN>DBASEMEM
```

will set it up for dBase. The only thing you need is the REBOOT command, which is given in Listing 3. (A machine-readable version will be found on the YC Bulletin Board).

The same technique can be applied to selecting one of multiple copies of AUTOEXEC.BAT. The only restriction is that REBOOT.EXE does not work if there is co-

```
main()
{
        int sysint();
        sysint(0x19,0,0);
}
```

Listing 3. REBOOT.C

resident software — like Sidekick — in memory.

Memory Drives

DOS 3.1 comes with a device driver — VDISK.SYS — which implements a memory virtual disk drive; in other words, it uses part of memory as a pseudo disk drive. Most multi-function and memory cards come with a similar piece of software. I'll refer to these memory drives as vdisks.

To the user, the vdisk appears to be a small hard disk. It is very fast — typically three times faster than a hard disk — and quite reliable (it has no moving parts). However, it has one major drawback — unlike a real disk, it is volatile; the contents are lost when the power goes off.

While the probability of that may be low, there is still the possibility of someone tripping over the power cord — that's not too bad, since you can always hit them. But when you finish work for the day and switch off, only to realise that you haven't copied the vdisk contents back to a real disk, the frustration reaches a high point.

To the user, the vdisk appears to be a small hard disk – it is very fast and quite reliable.

In order to avoid disaster, therefore, some guidelines should be followed for the use of vdisks:

1. Never, never put anything valuable into a vdisk. Live databases, for example, should not be placed in vdisk; however, database index files, which can be automatically recreated, are fair game.
2. Automate the operation of the vdisk by the use of a batch file.
3. Put the most frequently accessed files in the vdisk.

For example, in running dBase applications (a typical application where a vdisk makes a big difference) files should be handled as shown in Table 1. And the system should be automated with a batch file as shown in Listing 2 (assume hard disk C: and vdisk D:).

.DBF (database files)	left on hard disk (too valuable)
.NDX (index files)	possibly placed on vdisk
.PRG (command files)	definitely on vdisk
.FMT (screen format files)	definitely on vdisk
.FRM (report format files)	left on hard disk (slow anyway)
DBASE.COM	left on hard disk (accessed once only)
DBASEOVR.COM	definitely on vdisk (accessed very frequently)

Table 1.

```
ECHO OFF
ECHO Copying files, please wait
COPY C:*.PRG D:
COPY C:*.FRM D:
COPY C:DBASEOVR.COM D:
COPY C:*.NDX D:                 <- if room
D:
C:DBASE <application name>
COPY *.NDX C: /V                <- copy index files back
C:
```

Listing 4.

What Does It All Mean?

In this article, I hope I've shown you some of the power of batch files, some of the intricate things you can do with them. I hope I've also corrected the poor documentation from both Microsoft and IBM on this subject, and provided you with some useful tips and techniques. In addition, I've suggested ways in which the batch processing facility can be made more useful with the addition of some simple C or assembler utilities.

Where do you go from here? Undoubtedly, you'll have some problems of your own which are particularly amenable to batch file solutions. If you want to practice your batch file programming skills, here are some suggested applications:

- Menus,
- System configuration in response to user login,
- Electronic Mail,
- Security,
- Subdirectory clean-up,
- Location of overlay files, and
- Moving files between subdirectories.

Good luck with your batch file programming. If you come up with an interesting technique, share it with other YC readers by sending it to us at the magazine. I'll persuade Natalie to pay for contributions used, or even offer a prize for the best! □

Manipulating Batch Files

Here's a DOS Dirty Trick from Jeff Richards – a batch file created by a program to re-start itself.

HERE'S A DIRTY DOS trick that you might find handy: when DOS processes a batch file, it only reads as much of the file as it needs to execute the next command. It then remembers its position in the file, and returns there when the command or program has been executed. This means that it is possible for a progam to modify the batch file which invoked it, and to control the sequence of processing that will occur when it terminates.

Of course, for many procedures with a sequential processing path there is no need to fiddle with the batch file. Because one batch file can chain onto the next, a program that needs to control the chain of events simply creates its own batch file, and arranges for the batch file that invokes it to chain onto the file it creates. This is the way that installation programs work – SETUP.BAT might run SETUP.EXE then continue on to INSTALL.BAT – but INSTALL.BAT was actually created by SETUP.EXE based on the answers given to the installation questions. In itself, this hardly qualifies as a dirty DOS trick.

Circular batch file

However, a batch file created by a program to re-start itself is a rather dirty trick. Such a circular batch file might be useful when one program needs to run another program and retrieve an answer from it. Let's call the main program ASKER, the program that provides the response ANSWER and the batch file CONTROL. ASKER accepts a command-line argument of NUL which indicates that this is the first time it has been started, or a three-letter command returned from ANSWER.

Initially, CONTROL would contain the single line 'ASKER NUL'. When run, this would start ASKER. If the program needed an answer it would rewrite CONTROL to contain two additional lines – ANSWER and CONTROL and then terminate. DOS would execute ANSWER, which would work out the result of the question. It would then rewrite CONTROL from the start, replacing NUL with the reply (say, 'ONE'), and terminate.

DOS would then execute CONTROL, which is a batch file that starts ASKER with the argument ONE. ASKER would detect that this was not an initial startup, and would retrieve its answer. It should then re-write CONTROL to delete the last two lines, so that when it terminates, DOS detects the end of the batch file, and returns to the command prompt. (Actually, replacing the last two lines with an innocuous command such as CLS is a better habit.)

Each program that re-writes the batch file must be sure that the point in the file where processing resumes is unchanged. The contents of the file, both before and after this point, can be altered, but the byte position of the start of the next command line must remain the same.

This procedure is not restricted to cases where ANSWER actually returns a result – it could be that ASKER simply wanted to regain control after the job (such as a formatting a disk) was done. In this case it inserts the 'reply' into the first line of the file itself, so it can detect that control is being returned from a task that it initiated, rather than from a cold start.

Do-nothing batch file

If you have wondered why some software requires a tiny, do-nothing batch file to start it, the answer just might be that it will re-write that batch file if you ask it to execute a system utility, and then set it back to the standard form when it regains control. For circumstances where you don't want to, or can't, use DOS' CHAIN and SHELL facilities, rewriting batch files can be an effective dirty trick. □

HYPE ABOUT HYPERCARD, HYPERTEXT AND HYPERMEDIA

As an introduction to HyperCard, Stewart Fist waxes lyrical on hypertext – a flexible, programmable information retrieval system quite unlike anything that's been seen before!

THE COMPUTER world today probably has as many gurus as pop music and pop philosophy combined. These legendary computerised figures stalk the stages of high-tech conferences – destined evermore to be 'keynote speakers' sprouting futuristic philosophies and fantastic dreams.

One such figure recently re-emerged from his digital dungeons is Ted Nelson, the man who invented the terms 'hypertext' and 'hypermedia', and who authored *Literary Machines* trying to explain it all.

'We speak sequentially because we have only one vocal track and we write sequentially because books have numbered pages, but we don't think sequentially,' says Nelson.

It is the rigidity of database structures that he is attacking; the way in which we create records with dependent fields, and link them hierarchically into files.

Nelson envisages future databases as free floating packages of information (text, graphics, audio or video information) all interconnected by multi-dimensional links. His theory also involves the ability for each user to 'customise' their own database – in effect create their own links – so authoring tools are an essential part of hypertext.

Hypertext (or nowadays more properly 'hypermedia') concepts in the design of software would enable us to freely explore information in 'multiple parallel' paths, instead of being confined to a fixed path or structure, Nelson predicts. His Project Xanadu in the States has the rather fanciful aim of linking and cross-linking *all* of the world's information. You heard it right – *all*!

I met him three months ago in Seattle at the CD-ROM conference and we talked about hypertext for about an hour. But only on our second meeting, after a long hands-on session with OWL International Incorporated's CD-Guide hypertext system for CD-ROM based applications, was I able to get to grips with the concept.

I must admit that at first I thought Nelson was tending to confuse computer programming with some exotic Eastern religion or the Californian cult of EST. It is the philosophical abstractions that interest Nelson, not the nuts-and-bolts means of actually making computers work.

He is not the world's best explainer, and he deals in an area with difficult concepts and very few useful analogies. The other problem in understanding him is that hypertext is a 'normative direction' rather than a product or a design; it is a statement of aims about how things ought to be. The fact that these ideas have been around for over 20 years (since 1965 at least) is an indication of how difficult even the basic concepts are to achieve.

To get to grips with hypertext, you'll have to play with the software yourself – Melbourne-based Pica has managed to quietly import OWL's CD-Guide (for the Mac and IBM), and Apple is here with HyperCard. I promise you: these new products are the next wave of species software – like desktop publishing (DTP), expert systems and so on.

HyperCard is for the Macintosh, and Apple is swinging a lot of weight behind it. In fact Apple is bundling HyperCard software with every Mac sold, and if you've already spent your money on an old Mac, you can get a copy from an Apple dealer for under $100.

These two programs are only the first of a long line of future hypertext software – quite possibly in a few years these products will appear as primitive as VisiCalc does now. But they are very important products in the evolution of computers: like the cuckoo that heralds spring, I believe that HyperCard is the harbinger of 'informatics' with computers – as distinct from data processing.

I've been waxing pretty lyrically of late in these and other magazine pages about the imminent impact of CD-ROM and expert systems: now we have the third corner of the informatics triangle, a flexible, manipulatable, programmable information retrieval (pathway?) system quite unlike anything we've seen before. From today, the world is different.

CD-Guide

Let me give you some examples. First of all, OWL International's CD-Guide for the Macintosh and for the IBM PC (under Windows): this is a shell, much like the expert system shells, into which people put information on a subject. This information constitutes the 'database' – but it can be any type of information, it is not just limited to text and/or graphics.

CD-Guide allows readers to interac-

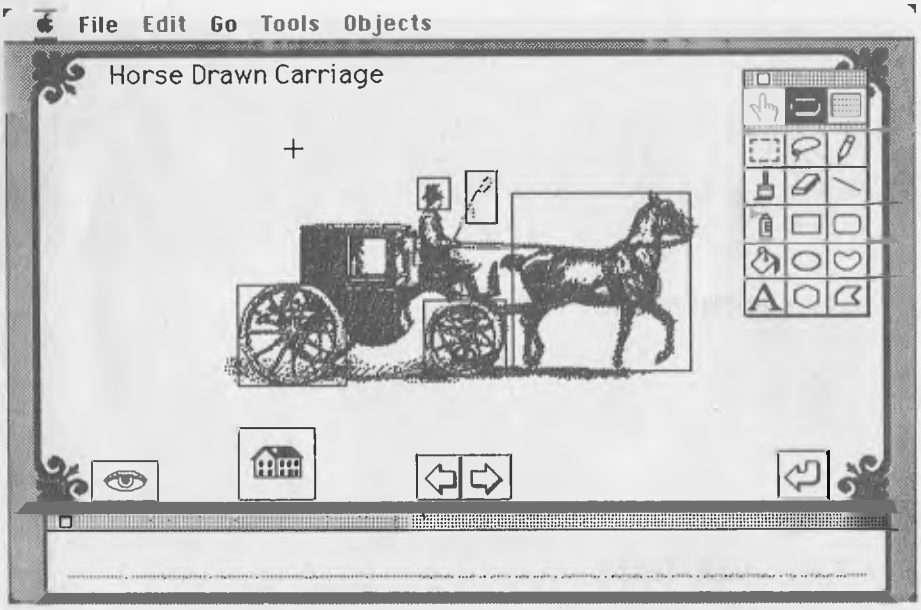


Figure 1. An illustration from a Hypercard demonstration stack showing the 'live' areas within an illustration. Clicking on the hat, in this case takes you through to where you get other types of hats.

tively explore information and allows them to 'tailor documents to their individual needs' by simply clicking on 'buttons' embedded in the documents. CD-Guide is a very early form of hypertext-type software but it does incorporate the powerful free form linking of text which is one of the basic concepts of the idea.

There are no commands; the entire user interface consists of either a one-button mouse (or a touch screen) so you can learn to use CD-Guide in a minute. You point and click on 'live' words, or mini-icons, or buttons, or objects within the screen to obtain the information you want.

CD-Guide grew out of a previous hypertext software product called Guide, which could be crudely and inadequately described as a cross between a word processor, a database and an idea processor. Its primary purpose is to present information interactively on the computer screen, but it can also be used to prepare printed documents.

The new software species of 'ideas processors' gets closer to hypertext than any other, I think. If you own ThinkTank or More, you'll understand what I mean. Each layer of the outline can have another layer behind, and these layers are directly related (tightly linked) to the layer above. If I click my mouse on a subject heading, that heading will 'open up' and reveal a series of sub-headings beneath – or at the final stages it will reveal some explanatory text.

Idea processors are essentially tree-structured. From the main heading, the tree branches into (makes connections with) sub-headings, sub-sub-headings and so on until it reaches the explanatory text at the leaf end.

True hypertext is not like this. It is a web-structure with all pieces of information having the same standing in the hierarchy, but with the ability to establish links of any kind, and to any degree.

For instance, suppose you were reading this text on a CD-Guide hypertext system – it would appear much as it does now. But if you were to pass your cursor over the text you would discover that certain parts of the text were 'live' – which simply means that the cursor would change to indicate that other data was linked to this word.

In the above paragraph the words 'CD-Guide', 'hypertext' and 'cursor' would probably be alive. If you were to click on these words, you would be instantly provided with other relevant information. In the case of 'cursor' it might only be a brief dictionary definition which would appear in the small window on the screen, but there could easily be another depth of explanation below this again which would take you into technical programming information about how cursors are controlled.

In this case the window would possibly have a selection menu which would let you choose whether you wanted to go into the programming data, or perhaps into a graphic screen illustrating different types of cursor styles, or whatever. The range of possibilities depends on the available information, and the links that you or the content provider have established – not on the limitations of the software.

If you clicked on 'CD-Guide' it might initially provide you only with the basic information about the programmers, OWL International Incorporated, with its address and telephone number. But you would probably also have an opportunity to pass through this window into one or more magazine reviews of the software; or lists of dealers; or a discussion on hypertext as a philosophy.

You might go down the yellow brick road of any of these pathways, clicking on live words as you go, and never get back to the original article – or you might go part of the way, then click on a 'return' button, and zap back to the paragraph you left.

The analogy here is probably very close to adventure games – although these tend to be rigid in structure with randomised sections. In fact, my first reaction on seeing Apple's HyperCard demonstration was just that; it provides the shell for a marvellous series of discovery games – so there is a lot of opportunity here for educational software writers.

HyperCard

Apple's HyperCard was rumoured for about five years – you may have heard it whispered about under the name Wild Card. Bill Atkinson the creator of MacPaint did the coding, and he is said to have spent four years just getting the user interface right.

HyperCard uses 'stacks' of 'cards' to store and sort information in a manner very similar to Xerox's Note Card program. It is obvious where the metaphor has come from, and the object-oriented control language also is derived from Small Talk.

Everyone in the reviewing business is having trouble describing HyperCard. See if these make sense: 'It is a multi-media database toolbox which can contain text, sound and graphics.' Or this, 'It can be best described as a non-linear way of tracing various aspects of a concept or feature, with multiple options at each step'.

Or Apple's press release, which says 'HyperCard is a software-based toolkit that gives users the power to use, customise and create new information using multiple information types such as text, graphics, video, music, voice and animation'.

I think you get more of an idea out of my just saying it is 'a serious adventure

game'. Remember that HyperCard, like CD-Guide, is a shell. It is an empty stack of cards into which you must put text, or graphics or even music.

If we are talking about CD-ROM as being the source of the information for that shell, then there will need to be a substantial input of text, graphics and information to fill the 600-odd megabytes of disk space. So HyperCard will probably establish a new information industry, which will be part-way between librarianship, computer programming, and expert-system type knowledge engineering.

Don't get the idea though, that HyperCard is dependent upon CD-ROM, or on encyclopaedia-type information. You can use the system to store and interconnect your own information – although I personally doubt whether people will go to the bother for mailing lists and suchlike after the novelty wears off.

I think HyperCard will find a place somewhat analogous to DTP within large companies (and as small service businesses), where a couple of people work together designing and constructing customised HyperCard applications. One thing is sure: HyperCard will eventually boost the sale of video digitisers and optical character readers (OCRs). These are the easy ways of handling large amounts of text and graphic information already in existence.

Let's get down to the nuts-and-bolts of HyperCard.

It consists of three disks plus a backup. There's a main HyperCard disk with several example 'stacks', a Help disk and a Stack Examples and Ideas disk. The main disk comes with a few desktop 'stacks' for an address file, a datebook, a 'to do' list, calendars and a filing system, while the Ideas stack gives you hundreds of stack templates, card designs and clip-art drawings. There is also a 225-page user tutorial and reference manual.

As a minimum, you'll need a 1 Mbyte Mac Plus with two 800 kilobyte floppies, although a hard-disk would be even better. The program itself takes up 368 Kbyte on disk, and there's not much point in using it unless you've got a reasonably large amount of data.

HyperCard can be used as a front-end to drive a CD-ROM disk unit or any other form of mass storage, and it will also have primitive communications – for instance the music system can be used for touch-tone dialing. Later versions will undoubtedly expand this capability.

I've seen a demonstration of a information package called Business Class which starts with a map of the world, then lets you move in progressively on any country, or city. Finally at the city level it gives you information such as hotels, exchange rates, cultural activities, transport, climate and so on.

```
8/4/87 10:58 AM                    Background Script

on idle
  if field "Time" is not the time
  then put the time into field "Time"
  repeat while the hour = it
    exit idle
  end repeat
  world
  put the hour into it
end idle

on openCard
  put the hour into it
end openCard

on ArrowKey left
  go next card in this background
end ArrowKey left

--on ArrowKey right
--  go previous card in this background
--end ArrowKey right
```

Figure 2. An example of the HyperTalk structured object-oriented language derived from SmallTalk.

There are a couple of calculator functions builtin to HyperCard, so you can change Fahrenheit to Centigrade. Obviously at a later time you will be able to go on-line to your bank and automatically update and do currency conversions.

As I see it, well-designed hypertext systems will become the central point of our new high-information world. Up until now we have been concentrating on data processing, and word processing – now we are into the age of Informatics with real information processing.

Hypertext systems will be able to access data on CD-ROM disks and update this from online links. They will also possibly act as the bridge between program modules that provide word processing, ideas processing, spreadsheet, and communication functions.

I doubt whether we will see elaborate hypertext systems for Apple IIs and the IBM PC range, mainly because you need the speed of the new 32-bit chips to make these intricate database management functions possible without excessive delays. We will also need the addressing range of the 32-bitters to provide the working space necessary for large databases – and without these hypertext is largely unnecessary.

Add to this the value of good quality graphics and multivoiced sound, and you can begin to predict where this is all going to happen in the PC world.

Obviously the Mac SE and Mac II are the desktop machines which hypertext will benefit the most – and Apple hasn't been slow to see the opportunities. But hypertext is also a system for mainframes and minis, and so a combination of PC terminals accessing on-line hypertext systems might be the way most of us will use hypertext. □

Product Details

Product: CD-Guide
From: OWL Incorporated
Distributor: Pica Pty Ltd, 38 Ardoch St, Essendon 3040 Vic.
(03) 370 3566

Product: HyperCard
From: Apple, 6 Rodborough Rd, Frenchs Forest 2086 NSW
(02) 952 8000
Price: HyperCard is bundled with the Macintosh range

Behind HyperCard - Part 1

THERE ARE QUITE a few computer journalists more enthusiastic than I am about HyperCard – and yet I think it's one hell of a program. Gareth Powell at the Sydney Morning Herald, for instance, promotes it as the most important program of the decade. I'm not sure that I would go out on a limb this far, but HyperCard is obviously revolutionary ... and also interesting. So even if the revolution fizzles, the fascination might remain.

One thing is for sure; you are not going to waste your time by learning about HyperCard. At the very least it is certainly the foretaste of programs to come. So if you're a Macintosh user with even the slightest interest in discovering how future data management programs will work, you should spend a few hours looking 'behind the scenes' of HyperCard and digging into its programming language HyperTalk.

This is such revolutionary software that some of the more showy features have tended to disguise important innovations that lie behind. For instance, HyperCard provides the novice with a gentle pathway into programming – an aspect which is quite unusual with substantial procedural languages. If you want to write anything worthwhile in C or Pascal or dBase III then you've got to sit down for two or three months and learn a mass of very esoteric commands and syntax *before* you can create code remotely worthwhile. Nine out of ten writers of Basic probably never write anything of any use to anyone. But HyperCard is different.

At the basic level, the program automatically programs itself through clicking and linking. It's not the first Mac program to do this, but it is the best. You can take this automated programming process a significant step further with very few lines of handwritten code – and you'll be capable of doing this by the time you finish this article.

HyperCard is undoubtedly revolutionary, but this extraordinary, complex procedural language is probably the easiest to program – a fact which disguises a number of important innovations!

This is what is so amazing – this extraordinarily complex 'language' is probably the easiest to program in a worthwhile way.

HyperTalk

There's a lot been written about HyperTalk as an 'object oriented' language like the famous SmallTalk, but it is not a true 'object orientated language', and it also owes plenty to the standard single-structure languages like dBase II.

Apple admits that HyperCard is not a 'classical object-oriented programming system, although it implements some object-oriented concepts, such as passing messages and having objects.' Are you any the wiser? Hang on – you will be!

Apple will soon have the *HyperCard Script Language Guide* in print if you really want to get deeply involved, but at the time of writing the manual is only available the draft level – and it has some mistakes. Danny Goodman's book, *The Complete HyperCard Handbook* still hasn't arrived at this time, although I hope to be reviewing it by the time this series is in print.

The only way to understand HyperCard is to play with it. And the only way to investigate HyperTalk is to go inside, have a look around and make some changes – so first of all make a copy of the disk HyperCard and Stacks and use that as a tutorial disk. Fire up the copy, and let's have a nosey around inside.

Clicking on the HyperCard icon should bring you up on the Home Card, and by clicking the left arrow at the bottom of the card you will go straight to the last card in the Home Stack – the User Preferences card.

Click on the Scripting radio button at the bottom – this provides you with 'tools' to enable you to create and alter stacks and script. Notice that the top menu bar has changed; along with the Tools menu there's now one labelled Objects. Why this emphasis on Objects?

HyperCard programs aren't flowing streams which start at one point and finish at another, rather they are collections of discrete code modules (called 'handlers'). Each module is part of a 'script' attached to an 'object'. There can be hundreds or thousands of handlers in a HyperCard stack, each of which performs its task quite independent of the others – although they communicate with each other by sending 'messages'.

The word 'object' is used in an analogous way. There is nothing physical about these objects. HyperCard has five different types of 'object' – Buttons, Fields, Cards, Backgrounds and Stacks – plus two quasi-objects (Home Stack, which is just a special form of stack, and the HyperCard program itself which often acts like an object).

You activate an object (usually, by clicking on it or sending it a command/message) and its handler code sends a message through a predefined pathway to another object. If this object's handler recognises the message some action is usually initiated.

For instance, go to the Home Card and click on the Phone icon. You jump instantly to the Phone Stack ready to insert a new phone number. Your click within the active area (a transparent button) over the Phone icon initiated a message from the attached handler. This message then flowed up through the various levels of the program until it was registered by another object (in this case HyperCard itself) and this triggered the switch to the Phone Stack.

The action has to be predictable – in other words there must be some order to the sequence of objects, and this is where the heirarchy comes in. Messages flow upwards from low-level objects (buttons and fields) to higher levels – and the higher the level the more widespread the action.

Obviously the handler that actually made the switch between Home and Phone stacks needs to be at a higher hierarchical level, since its effect (the switch to a new stack) is felt by all the cards, backgrounds, buttons and fields below it.

Think of it as an inverted tree structure with one Home Stack at the top trunk level, a number of Stacks below on main branches, then many Cards each with Background, Fields and Buttons. It's a matter of one Supreme Commander, a few Generals, many Lieutenants, hundreds of Corporals and thousands of Privates.

Change back to the Home Card and we'll have a look at the the script that initiated the action that swapped you to the Phone stack. From the Tools menu, select the Button tool in the middle of the top row. Then click on the Phone icon to select this button (you'll see a rectangle of walking ants), then go to the Button Info bar of the Object menu and click again.

In the dialog box which now appears, click on the Script panel, and this will reveal the handler responsible for the action. The script attached to this button only has one handler that begins at the keyword 'on' and finishes with the line beginning with 'end' –

```
on mouseUp
   go to stack "Phone"
      visual effect zoom
end mouseUp
```

It couldn't be much simpler, could it? It doesn't take a college education to realise that 'mouseUp' is the release phase of clicking the mouse, and that 'go to 'Phone'' is the message (the command) that causes the stacks to swap. This issued message has the messageName 'go', (which is also a HyperTalk command) and so in this case it is captured and used by the HyperCard program itself (but it need not be). 'Phone' is the parameter. The word 'to' and the quote signs are superfluous, and are included to make the script more readable and avoid ambiguity.

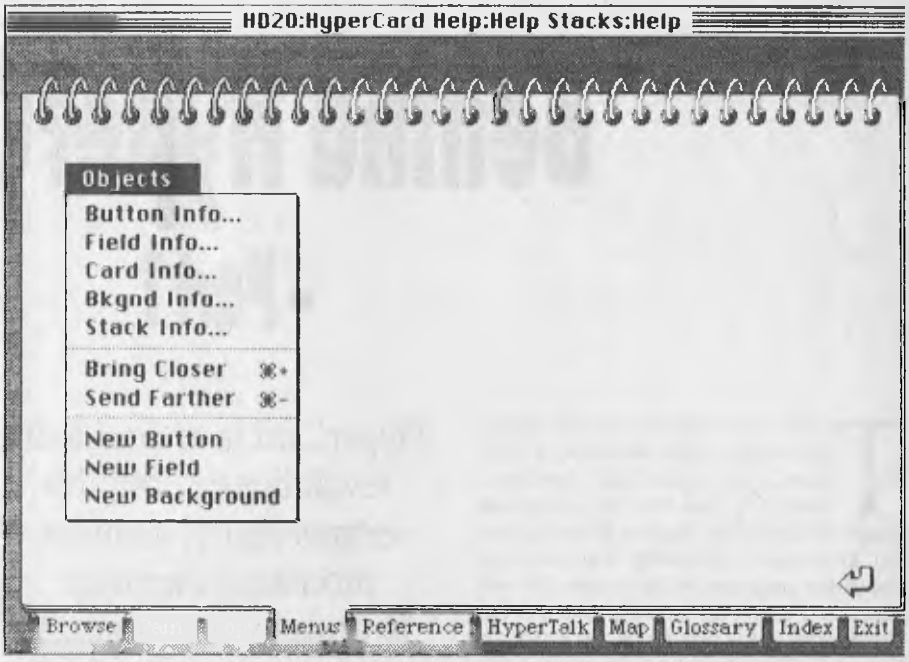


Figure 1. *In HyperCard, not all objects are equal. There's a rough sort of heirarchy in the status of objects – Buttons and Fields are at the lowest level, then progressively Cards, Backgrounds, Stacks, the Home Stack, with HyperCard itself at the top. The Home Stack is a special case, and you can treat HyperCard itself as an 'object' since it can both send and receive messages.*

There's also a line of code here which creates the visual zoom effect, but it's not necessary for the system to work. It is, however, a good indicator of how English-like HyperCard's command language is, and this makes it easy to learn. In fact, we can do some programming ourselves while we are here.

Notice that the cursor assumes the I-beam style (as in MacWrite) whenever it is in the script area. This shows that you can add, modify or delete command lines in exactly the same way as you would change the text of a letter.

Try it now. After the 'on mouseUp' line, insert a new command line that just says 'beep' (no quotes) or 'beep 3', if you want to be flashy. Click on the OK panel, de-select the button icon in the Tools menu and try it out. From now on the Phone icon, and only the Phone icon, will sound the bell (once or three times) when you select it. See how easy programming is!

While you are at the Home Card, use the Objects menu to have a look at the script associated with the Home Card itself, its background and the Home Stack. You'll see an increasing degree of complexity as you get further up the heirarchy, as you would expect.

At the Card level there are three script modules ('handlers') to do with updating the time on the card and showing or hiding the Copyright notice. The Copyright notice only appears at boot-up.

The Background script has been set to handle things like the non-appearance of the message box and the menu bar when the Home stack is opened, and the Stack script is primarily concerned with external commands ('XCMD' in the jargon – from C, Pascal or the Mac assembly language) and the storing of information such as the User's Name and preference level.

But, just as the higher levels are obviously hard to understand, the lower levels are certainly easier than you expected, aren't they? You might not be able to reprogram a Home Stack in your first few days, but you can manipulate buttons and fields, and there's a progressive learning path through this program.

Heirarchy

In HyperCard, not all objects are equal. There's a rough sort of heirarchy in the status of objects which you will see if you pull down the Objects menu (Figure 1). HyperCard recognises, in order of importance, Button and Field objects as equal bottom, then progressively Cards, Backgrounds and Stacks. The menu list is upside down to my way of thinking; Stacks should be on the top.

The full extent of the operational heirarchy of the program has Buttons and Fields are at the lowest level, then progressively Cards, Backgrounds, Stacks, the Home Stack, with HyperCard itself at the top. The Home Stack is a special case, and you can treat HyperCard itself as an 'object' since it can both send and receive messages. You will understand the significance of this later.

Think of the program as a five-storied building with a basement. When messages arrive at the front door they are first checked at the lowest level to see whether they are intended for this floor. If they are not intercepted here they are passed on to the next floor, then to the next, and so on until some may finally trickle through to the big boss at the top and he deals primarily with 'commands'.

Mouse messages come through the sewer, so these are first checked at the basement level (Buttons and Fields), while most other messages come in at the street-level (Cards). But however they enter the building, messages always move upwards, if they are not intercepted.

The internal activity of HyperCard all depends on 'messages'. For instance if you click down on the mouse button this action sends a system message called 'mouseDown'. When you let the button up, the message 'mouseUp' is broadcast. If you don't do anything at all, the program constantly broadcasts the message 'idle'.

The term 'message' here is used in a wider fashion than the term 'command' (builtin keywords initiate some action). It often includes both a command and a parameter, such as 'go' (the command) 'Home' (the parameter). It also includes system-event messages sent as a result of some action (like clicking the mouse), 'function calls' (builtin calculations which return some value to the initiating handler), and message or function names that you have made up during scripting to trigger some action further up the object heirarchy. Many messages have parameters attached, but they don't need to. Here are some examples of messages –

HyperTalk

HYPERTALK IS A very flexible language. It doesn't worry about upper or lower case, nor does it have an excessively strict syntax. However, to avoid ambiguity it is best (but not essential) to –

a) Enclose names within quotes (for example, go to stack 'Home').

b) Put the word 'stack' in front of a stackName,

c) Put the word 'card' in front of a cardName/number,

d) Address a card, button or field by its ID number,

e) Put the words 'card field' in front of a card field name to avoid confusion with a background field, and

f) Ideally refer to an object by both its type (for example, 'card') and identification number (for example, ID 5734)

go Home – HyperTalk command + parameter

mouseUp – system 'event' message

xyz – message to author-defined handler that will start with 'on xyz'

average (Len1, Len2) – function call + parameters

nameLength (lastName) – author-defined function call + parameter.

MouseUp or mouseDown (event) messages flow first to the topmost button or field attached to the area under the cursor (they are directed to the appropriate button or field by the program, since it keeps track of where the cursor is).

The system-event messages mouseEnter, mouseWithin, and mouseLeave are generated within the program and flow to the appropriate button or field object. But these are post-facto (after the event) messages, and they are sent after the program has already decided which field or button the cursor has entered/left.

The mouseLeave message is sent only to the button or field that the cursor has vacated – and then it passes on up the heirarchy until it finds a handler that needs a 'mouseLeave' message.

If they aren't recognised at the Button or Field level, all messages pass to the appropriate Card, then to the Background, the Stack, the Home Stack and on the HyperCard. Messages can be intercepted at any level where the appropriate script handler exists, and they can act from any level.

For instance, a handler which puts the time into a card field, doesn't have to be at the field level, it can be at the Card, Background or Stack level just as easily – and at these levels its influence will be progressively wider.

When you are programming, it is usually your decision as to which level you will write a handler – and you decide this on the basis of how wide you want the influence of the handler to be felt.

System messages aren't the only kind. As we have seen there are also HyperTalk command statements, function calls, and also you can type messages into the message box by pulling down the Go menu. This is actually a form of direct programming.

Try writing 'go to next card' in the message box. You'll see that the reaction is exactly the same as if you'd clicked on the appropriate arrow/button – because the issued message was exactly the same.

Your keyboard-entered message came in at the second (Card) level, then travelled up through the system until it reached HyperCard itself. At this level the 'go' message was recognised as a command and the 'next card' parameter was acted upon.

If you select the Stack Info bar from the Objects menu, then show its script, and write a short useless handler like –

```
on go
    play "boing"
end go
```

You will find that this short-circuits any messages in the message box like 'go to next card'. If you've got card or background arrow buttons that previously took you to the next card, or the last card, you'll find that they don't work either. Everytime you try, the system just 'boings'.

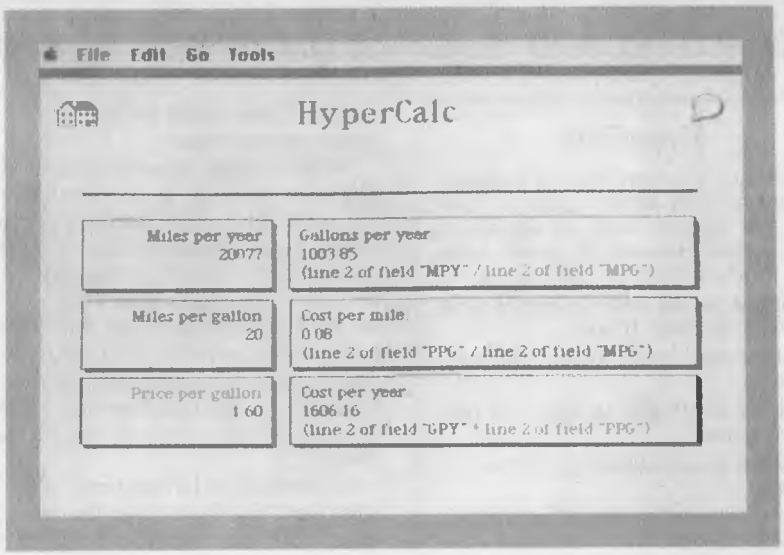
Taking control

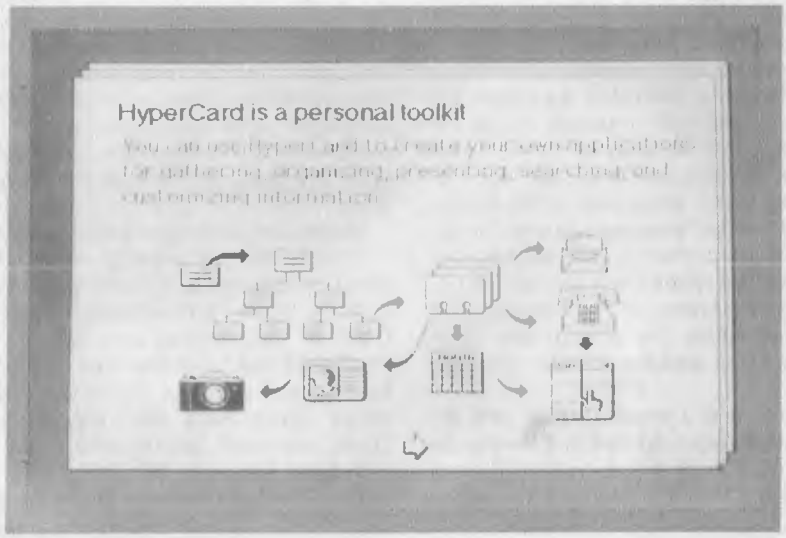
This is a trivial example with important consequences, because it means that you can write your own handlers using a messageName identical to a HyperTalk command, and steal control from this command. You can use the command for your own purposes, then pass it on again, if you wish.

But we are jumping ahead too quickly!

Objects don't necessarily need to have scripts associated with them. For instance, from the Home Card, select the Field tools (right in Tools menu) then click on the small field that holds the Time on the bottom right of the card. Go to the Objects menu, select Field Info, then click on Script and you'll see that this 'field' object doesn't have its own script.

This is natural enough; this is a special





field just for display. Clicking on the time or changing the numbers is not intended to initiate any action through the system.

This object (a field) is a passive receiver of messages originated by the hardware clock chip. However there must be a script handler somewhere to intercept the clock messages and cause the numbers to be displayed – and you'll find this in the Home Card script. It could have been higher at the Home Stack level.

So some objects have no script, and can therefore send no messages. At the other end of the spectrum, some parts of the HyperCard system are constantly sending messages, and other objects are constantly receiving them and interpreting them through script handlers.

If you are moving the cursor about on the screen and you enter an area allocated to a button (or a field), HyperCard sends the message 'mouseEnter' through the system (entering at the appropriate button or field) and cycling back to itself. While the cursor remains within this hotspot, the program constantly broadcasts 'mouseWithin' messages, and as it leaves the area it sends 'mouseLeave'

The button field itself usually doesn't do anything with this stream of cursor-position messages because its handler has been set to look for the mouseUp command of a click unless it has been deliberately changed – which is what we will now do!

For no other reason than to get comfortable with modifying scripts, find your way again back to the script handler belonging to the Phone button on the Home Stack, and change the two 'mouseUp' references to 'mouseLeave'. Exit the script, re-set Tools, then pass your cursor over the Phone icon. Your change now activates the stack swap as you withdraw the cursor from the button area; you don't need to click.

Here's another fun one. Select the Time field on the Home Card, and enter the three line script –

```
on mouseWithin

beep

end mouseWithin
```

Now whenever your cursor is within the Time field area on the bottom left of the card, the bell will sound.

Customisation at last!

In Part 2 we'll look further at messages and handlers, and get a better idea of the way different messages are handled within the system, and what this means to the programmer. □

Behind HyperCard - Part 2

IN PART 1, We saw that messages were read and acted upon by message-and function-handlers which created some activity or, in turn, passed a new message through the system, or were acted upon by HyperCard itself. Let's create a new stack and see how much script is automatically generated.

First, go to the New Stack bar in the File menu and create a stack called Trial. Don't forget to knock out the check box which says 'Copy current background' – we don't want a background at this stage.

Now let's create a background ourselves. Go to any card in the present demonstration stacks. Virtually all of them have a Home icon that we can copy by choosing the Button tools, selecting, then copying the button. Jump back to the first card in our new stack, select the Edit menu, set the Background to ON, then paste our Home icon into the background.

You can now add a couple of your own fields and decorate it with a few MacPaint-type boxes or lines – whatever takes your fancy. Now, after switching off the background mode, you should create a few new cards so that you have something to play with.

At this stage, have a quick look through the scripts in the object hierarchy of the Trial stack. You'll find nothing in any of them (Stack, Background, Card or Field) except for the Home button which contains a three or four line 'handler' which is waiting for a 'mouseUp' message, at which time it will transmit a 'go to Home' message.

This lack of script handlers is what you would expect, if you think about it. Apart from the Home button, we haven't created anything that initiates an action, or awaits a message. Everything else we can do with this stack needs to be done through the menu bar, and these create commands that are all handled by the top management level of HyperCard itself.

How different messages (and functions) are handled within a HyperCard system – and what this means to programmers . . .

Although there is no script attached to most of these new objects (stack, backgrounds, cards, fields and buttons), they have each been provided with an identity – even if you didn't bother to name them. If you pull down any of the Card Information boxes from the Objects menu, you will find both a stack-order number, and an ID #. If you add or delete cards from the stack, the order number can change, but once the ID is allocated it remains the same for the life of the program.

Time field

Let's create a Time field in the same way as we did the Home button. Go to the Home Card, select the Field tool, click on the Time field area at the bottom left of the card to select it, then copy this field through the Edit menu. Back to our new stack then paste in our Time field and try it.

Hisses and boos! Nothing works. There's no time in the field. Why? If you go back to the Time field on the Home Card to look at its script you'll find that there isn't any. The messages labeled 'time' come into the system at the second-floor level.

Remember: only the mouse messages (and a few special cases) arrive at the lowest Buttons and Fields level of the program, therefore if you are going to intercept 'time' messages from the clock, the script has to be at the Card level at least. (Note: 'time' is a function)

If you jump up to the Home Card script you will find the right handler there –

```
on idle
 put the time into card field "Time"
 pass idle
end idle
```

If you can remember back to Part 1, I explained that the 'idle' message was constantly being sent by the system when it was in a wait-state. So this 'handler' is saying: 'Since we've got nothing better to do, put the time message up on the screen inside the card's Time field. Then pass on the idle message (some other part of the system may need it), then end this handler.'

It is worth reiterating that this handler doesn't need to be at the field level – even though its effect is to place a variable in the field. The handler doesn't even need to be at the Card level – it could just as easily be at the Background or Stack levels.

So let's go back to our new background and type this handler into our new Trial stack. If we put it into the Card information, then the time will only come up on that one card but we want the time to be on every card in the stack (this is why we pasted our field into the background – rather than the card).

So we will copy the handler into the Background script making one modification. The second line must now read:

```
put the time into background field "Time"
```

cows come home, but nothing will happen unless you are on a button, a field, or a menu area – in which case the message will probably be passed to a handler in a script at the appropriate level.

HyperCard knows which button to direct the mouseUp message to, because it has some sort of a cursor-location/message-direction functional unit built-in at the top level. This unit also issues a series of cursor position messages all the time you are moving the mouse.

As your cursor entered a field or button area, it issued a 'mouseEnter' message, then a constant stream of 'mouseWithin' messages, and finally when you leave the area, a 'mouseLeave' message.

Target

The term 'target' in HyperCard has a special meaning. If a message is sent to a button then this is the 'target', even if the message is not interpreted by the button's script but passed on to a handler at a higher level in the hierarchy.

'Target' is a function that remembers the entry point into the hierarchy. Try this handler at Background or Stack level:

```
on mouseWithin
 put the target
end mouseWithin
```

The word 'the' in the second line is mandatory unless you add parentheses after the word 'target' – as in 'put target()'

'Target' is a function, and functions are identified either by the word 'the' before the function-name, or '()' after. If there are any parameters they go between the parentheses, in order, and separated by commas.

In this handler we didn't need to say 'in the message box' at the end of line two since this is assumed (it's the default) if nothing is specified. But for clarity and consistency sake we should probably make this line:

```
put target() in message box
```

Bypassing

As you can see the hierarchy of HyperCard is all-important, however you can bypass the strict order in a couple of ways. The 'send' command short-circuits the system by sending messages directly.

For instance if you are at Card 3 in a stack and the handler issues a message line:

```
send "mouseUp" to button 3 or card 15
```

– the message will travel there directly, bypassing any 'mouseUp' handlers in between.

Let's get back to some practical examples. Create a button on a new card in your Trial stack and link it to the Home Card, then modify the script to read:

```
on mouseUp
  send "go to Home" to HyperCard
end mouseUp
```

You will see that the button still acts the same as if you'd left the middle line:

```
go to Home
```

All that 'send' command has done is to leap-frog all intervening stages and issue the command directly to HyperCard at the top of the hierarchy. There is, however, a difference between the two handlers. With the 'send' version you can no longer intercept the 'go' message by placing another 'go' handler at a higher level.

Try this line between mouseUp handlers:

```
send "go to Home" to stack "Phone"
```

– then watch the top window bar (which shows the stack name) when you click on this button. It jumps to the Phone stack first, then to the Home.

The concept of the hierarchy of objects that I illustrated in Part I with a five-storey building plus a basement (for Buttons and Fields) is actually a bit too simplistic. Apple themselves have also been carried away with over-simplifying the concepts, with the result that some of the explanatory material in the draft manual is misleading and sometimes downright incorrect, so be warned!

Let's write a series of simple handlers to check out the hierarchy of HyperCard. You should know by now what would happen if you created a button and wrote in the script:

```
on mouseUp
 go next card
end mouseUp
```

If you clicked on this button you would swap to the next card. Now write the same script in at the Card level.

Apple's manual says that if you now click on any button youwill jump to the next card – but, of course, that is only true if the button doesn't have any mouseUp handlers attached – and most of them do. Clicking on a button is what buttons are primarily for, and so buttons usually have a mouseUp handler of some sort – and very rarely will these have the line 'pass mouseUp'.

So, try clicking outside a button. It sometimes works, and sometimes doesn't. If you are on an area of the card or background not covered by a field or button, then your click will take you to the next card. But if the cursor is within a field, there will be no action, and if it is within a button area with a mouseUp script attached, then this action takes precedence.

This illustrates a very important point. When you think about it, the mouseUp message isn't being generated by the mouse – except in the mechanical sense. The message comes from within the HyperCard program, and it must flow through a couple of secondary stages in the program in order to be directed to a particular button or field.

Somewhere within the program (at the highest level of our hierarchy) there is a functional unit of code which is checking the position of the cursor, and recording whether the pointer is within, or outside, screen areas which delineate fields, buttons and menus.

When you click in a field area, the program allows you to begin to write text within that area. So the mouseUp message must be intercepted by a top level handler within HyperCard itself, and it is this signal that causes the pointer to change to an I-beam in preparation for entering text. This handler does not pass the mouseUp message through, so it can't be used to activate a handler at the Card level.

The same applies whether you add the above handler to the background or to the stack. Clicking within a button area means that the mouseUp message is intercepted by the button script (in most cases) while, within a field or menu area, it is intercepted by the HyperCard program itself.

You'll now find that the clock works perfectly, and it'll appear on every new card in the stack, as long as that card uses the same background (with HyperCard, it doesn't have to!).

Now let's get tricky! If you select and copy this Time field again, then paste it into one of the Cards in the stack at the Card level, you can get two clocks working on the same screen – one on the Background and the other on the Card. You'll need to add an (almost duplicate) handler to the Card script with the second line rewritten to:

```
put the time into card field "Time"
```

Note that on this occasion we specify 'card' field.

If you don't specify either card or background, then background is assumed for fields (it's vice versa for buttons). Note also that we now have two quite different fields called Time, which is not wise, but the system can handle the potential confusion as long as they exist on different levels. Much better practice would be to rewrite the second handler lines to direct the time variable to card/background field ID *xxxx* – then there can be no ambiguity.

Flexibility

Hypertalk is a very flexible language. There are many superfluous words often included in HyperCard scripts simply to help with clarity. There are also alternatives and condensed versions of commands and keywords.

For instance 'go' and 'go to' are the same, and you can say 'go home' or 'go to stack "Home"', it makes little difference – although enclosing the name of the stack in quotation marks ensures that it won't be confused with some variable, or whatever, that might exist with the same name.

HyperCard treats upper and lower case the same but, by convention, you should use uppercase to make joined words more readable (like: mouseUp).

You can also address an object in a variety of ways. For instance you can name a card "Mabel" if you want to, and then issue the command 'go to card "Mabel"'. Or you can specify a card as the 'second card' in the stack, or as 'card w', or as 'card ID 2439'.

But I digress. Back to our example. The third line in the clock handler was:

```
pass idle
```

– in both the Card and the Background script. Go into your Background script and delete this line. You'll find it makes no difference. Now delete the line from the Card script. You will find that the Background clock stalls. It can't increment because the message 'idle' is being intercepted at the lower Card level, and not being passed on.

This is a very graphic illustration of the way HyperCard's object hierarchy works – and how messages are absorbed and used, and/or passed on through the system to higher levels.

Incidentally, I am not recommending that you put a time field into every HyperCard background. It's OK if your stack is going to be read-only, but if you attempt to write to a field on a card with such a time-field already in place, you'll find that your cursor suddenly disappears once every minute.

What happens is, that the time-field takes control of the cursor automatically to insert the updated time, and you will have to laboriously reposition the cursor and click to get it back into the writing area each time.

Modular

Within an object's script, messages are also handed down on a modular basis. The message is tested against the first handler, then the second, then the third . . . and so on. Each handler consists of a number of discrete elements – usually automatically indented in the script to make them easier to read.

The most basic form of handler is three lines of code like:

```
on mouseUp
  go to stack "Home"
end mouseUp
```

The first line must begin with either the keyword 'on' which identifies a message handler, or 'function' which identifies a function handler – so these lines are called 'message indicators'.

Functions are inbuilt mini-programs which can calculate averages, extract characters from a string, and so on – they always return some value to the handler which called them. We'll look at functions later, here we are dealing only with message handlers.

The last line in the structure must begin with the keyword 'end' (end-of-handler indicator) and be followed by the handler name (here, mouseUp) which must be the same as the handler name on the first line. These are the 'book-ends'!

When a message reaches an object it is checked against the handler(s) in the script, and if the message-name matches a handler-name, the lines following are executed in order until the 'end (handler-name)' line is reached.

There can be any number of execution lines between the 'on *xxx*' and the 'end *xxx*' lines, but it makes sense to keep them short, if possible. You can always call another handler if need be – just like a GOSUB in Basic.

Let's look at our simple example above under the microscope. The message 'mouseUp' comes into the system at the Button/Field level, directed at a particular button (dependent on the cursor position). Since the message-name 'mouseUp' matches the handler-name in the first line of the button's handler ('on mouseUp'), the subsequent line(s) are executed and the command statement ('go to stack "Home"') is issued through the system as a message-in-reply.

The original message ('mouseUp') doesn't go any further in the system unless the special statement line 'pass mouseUp' is included in the handler.

Normally, if none of the objects in the chain have a handler-name that matches the message-name, the message will flow right on up to the top level, which is HyperCard itself. Hypertalk has a whole series of command interpreters which are virtually built-in handlers, and these initiate action at the top program level.

But not all system messages or commands have a handler within Hypertalk itself. For instance mouseUp doesn't mean a thing at the Hypertalk level (unless the cursor is within a field) – which is why you can click your mouse button until the

Names

Don't get carried away by the need to use pre-defined terms like mouseUp or startUp as handler names. Obviously a standardised set of words has been established so that command routines can be built into the program and so that keyboard and mouse events have a fixed nomenclature, but when you are writing your own script you can use any handler name you want – including those already used as commands and in-built functions. Just remember the hierarchy!

Here's an example. In a button script you could write a handler:

```
on mouseUp
 xyz
end mouseUp
```

Then higher in the hierarchy (say, Stack level) you could have:

```
on xyz
 go next card
end xyz
```

It works perfectly. The mouseUp message flows down from HyperCard and causes an 'xyz' message to be sent from the Button level. The message flows up through the hierarchy and, in turn, causes a 'go' message to be sent (together with its parameter 'next card') from the Stack level. This triggers the action at the HyperCard level. It's actually a circular flow of control.

You'll find a lot of these author-invented messageNames used in Hypertalk: words like 'getHomeInfo' which are descriptive of the job they are being called upon to do. You aren't limited only to the set messageNames specified by the program's creators.

When a non-HyperTalk messageName (like 'xyz' or 'findNextCard') is used in a handler line without a parameter it simply calls another handler into action, somewhere higher in the program, as if it were a sub-routine.

This handler would be executed, and if it triggered another handler into action, it too would be executed. Finally, after the chain of events had come to a stop, the control would revert to the original handler, and the next line in the handler would then execute, in turn.

There's an exception here. Any line beginning with 'pass' must be the last line before the 'end' line. Once the original message has been 'passed', the remainder of the handler ceases to function.

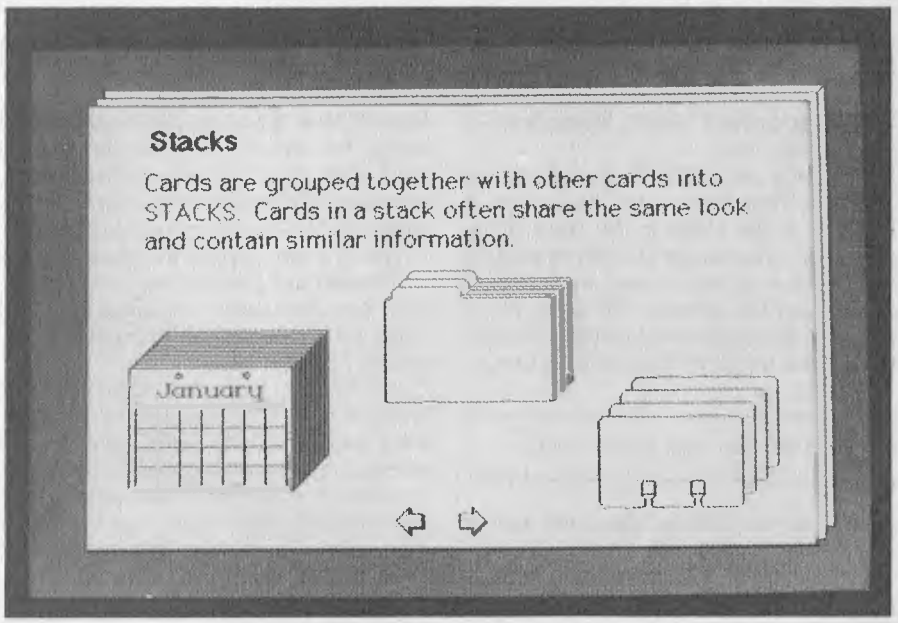
In the example above, the execution of the program would move down the Button script, then from the xyz line it would effectively GOSUB to the Stack level handler which issues the 'go' message, then to the HyperCard level to make the change, then return to the 'end mouseUp' line of the Button handler.

The chain of handler actions could, in fact, jump a number of stages further away and still return. Handlers can call other handlers, which call other handlers – but control will always return to the source of the sub-routines to complete the original handler before the regular pattern of flow through the object hierarchy is re-established.

With most sub-routines these secondary handlers will usually be in the same script, and there's one slight oddity here. The message being sent from the Button (or any other object) flows, first of all, through its own script – from the beginning – before traveling on up the hierarchy.

So it doesn't matter whether the secondary (sub-routine) handler appears in the script above, or below, the primary handler. The message will still find the handler and create any action, then execution will return to the next line of the primary handler.

There's a danger here in that you can turn messages back on the handler in an infinitely oscillation – like loudspeakers feedback through a politician's microphone. Recursion is possible, but it must be used with discretion. □



HyperCard commands to try

Sandwich these commands between mouseUp or mouseEnter lines and see what they do. To automatically create a dialog box, then add any input to a variable called 'It', try this –

```
Answer "question"
Answer "question" with reply1 or reply2
Ask "question"
Ask "question" with defaultAnswer
```

Here's a handy routine –

```
Open file fileName
Close file fileName
```

Using this command, HyperCard will get whatever is in source (a field, for example) then execute it as a command –

```
Do source
```

For the equivalent of a mouse selection –

```
DoMenu menuItem
```

It's also worthwhile experimenting with: find, get, go, hide, show, play, put, read, write, send, set, sort, type, and wait. And, experiment with the keyboard: ArrowKey (up, down, left, right), EnterKey, FunctionKey, ReturnKey, TabKey; and the CommandKey, the OptionKey, and the ShiftKey – these last three are functions and therefore return values which can be put in message box, or used in some other way.

Behind HyperCard - Part 3

Serious users may see HyperCard as a junior sibling to 'real' object-oriented languages like Lisp or SmallTalk – but for non-professional programmer's the language is superb!

THIS IS THE last in this brief four part introduction to HyperTalk, the language of HyperCard. We have not aimed to produce a definitive tutorial – more a taster, to give you a flavour of the language, and to demonstrate how easily simple, useful scripts can be added to your stacks.

If you want to go further with this language, we suggest you buy one of the many good books now available on the subject. Danny Goodman's *The Complete HyperCard Handbook* is excellent for the beginner, as is Carol Kaehler's *HyperCard Power*. For more advanced programming try Dan Shafer's *HyperCard Programming*.

Despite the obvious enthusiasm of a lot of Mac programmers for HyperTalk, it is becoming increasingly obvious that the programming language has substantial limitations. Many serious programmers see it only as a junior sibling of 'real' object oriented languages like SmallTalk and Lisp.

But for the non-professional user/programmer the language is superb. It has a complete set of system messages (mouseUp and so on), excellent mathematical functions and text-string commands, variables (both global and local) and even includes nestable condition testing (if ... then).

And the language is dynamic; the new release (Version 1.2) fills some of the gaps and improves the range of possibilities without creating any problems with past scripts. This latest version has just arrived on my desk, so it should be in the hands of dealers by the time you read this.

Some changes

One of the more important changes is to increase the range of actions of 'me' and 'target'. We discussed 'target' briefly in the previous part: remember, it is a function that returns the point of entry for a system message. For instance, if you click on a certain card field or button, then the button or field ID or name is returned and can be used or displayed as required. If the cursor is outside a button or field when you click, then the Card ID is returned.

Try this simple script to understand how this works. We'll use the mouseEnter, rather than the mouseDown or mouseUp system messages, and add the script at the card level –

```
on mouseEnter
   put the target
end mouseEnter
```

MouseEnter is a system message that is sent whenever the mouse enters an active area (field or button), and if we don't specify where to 'put the target' information, then it will be displayed in a message box by default.

Previously 'the target' referred only to the object (the button or field), and if the target was a field, there was no way of accessing its contents. In Version 1.2, 'target' (without the 'the') refers to the contents of a field. Therefore –

```
put target into myVar
```

or

```
put "Fred" into target
```

are now acceptable, whereas they wouldn't have been in the earlier versions.

It is difficult to separate the concept of the object and its contents sometimes – but this distinction needs to be made if you are to keep the logic of the language in your handlers.

There is a similar problem with the use of 'me'. Change the middle line of the above handler so that it reads –

```
on mouseUp
  put the name of me
end mouseUp
```

You will find now that whenever you enter a field or button area, the message box will pop up with the name (or ID number) of the Card (not the button or field). The term 'me' always refers to the object containing the current handler – and since you wrote this handler at the card level, it will always return the name or ID of the card. Try this change –

```
put the name of this background
```

and you will see that the word 'this' is also an object descriptor.

To get this all straight: 'target' will return the name or ID of the entry point for system message; 'me' returns the same for the object which holds the current handler; 'this' returns the current card, background or stack (not field or button – and you must specify).

After you've played around with these changes for a while, try this – but only if you have the new Version 1.2. Firstly, create a new card field and give it the name Display; then at card level write –

```
on mouseUp
   put the target into me
   put me into card field Display
end mouseUp
```

This handler illustrates the newly extended operation of 'me'. Previously 'me' was just a convenient synonym for the current object, but now it is a 'container' as well. In HyperCard jargon, the term 'container' means 'variable' in its widest sense.

You might think this distinction is a bit esoteric, but HyperCard fields are also both containers and objects. The term object refers to the status in the message hierarchy, while container indicates that data (text) can be added to the fields, and that the fields will hold and regurgitate this data when required. A field is just like a big variable that is visible on the screen. Handler lines like –

```
put the name of this card into card field Display
```

are virtually the same as –

```
put the name of this card into myCard
```

In the first case the card filed 'Display' must be created first before the handler is activated. It doesn't need a name because an ID number is automatically generated. In the second case, the new variable 'myCard' is automatically created to hold the information.

You assign value to a field by typing into it, and you assign value to a variable with a script line that begins with the 'put' command. You don't need to establish local variable names before you use them; but note the qualification 'local'.

HyperCard has both local and global variables. The distinction is between those which are only valid during the execution of the current handler, and those which carry the value throughout the program. Note that a local variable is valid only within the handler – it doesn't extend to other handlers in the same script –

```
on mouseEnter
   put the target into myVar
end mouseEnter

on mouseWithin
   put myVar
end mouseWithin
```

will not give you a message box revealing the name of the target – it will simply print the word 'myVar' inside the message box every time. The middle line of the second handler needs to be the third line of the first handler (and the second handler deleted) for the system to work, since myVar hasn't been defined. Therefore, by default, it is a local variable. You could also make the above script work by adding a second line to both handlers –

```
global myVar
```

which is the way you define global variables, before the 'put' statements. You must create the global definition both in a handler before adding a value, and also before using that value in another handler – which will seem a bit odd to most programmers.

You can declare more than one global variable in the same line, with the names separated by commas, if you wish –

```
global chapters, sections, pages
```

defines three different global variables into values can then be put.

If you change the value of a global variable anywhere, you change it everywhere – but the changes aren't saved to disk between sessions or when you swap out of a stack.

A good example of a built-in local variable is one named 'it' which is used by default if no other variable name is declared. So if you write the line –

```
get the ID of target
```

the target ID will be written into a variable called 'it'. You haven't had to define it – or (in this case only), even use the variable name, but the next line –

```
put it
```

will result in the target ID being revealed in the message box.

Scripts

The use of all these common English words and defaults makes HyperTalk appear to be deceptively simple to write, but in some ways it also complicates the reading of scripts. It is hard to get use to the idea that 'it' and 'this' are active parts of the language, not just loose connective words.

It takes some familiarity with the program to remember that 'put it' means 'place the contents of a variable (which has probably not even been named, let alone defined) into the contents of a message box (that isn't mentioned) and display this on the screen (which is assumed).' One rather difficult distinction the tyro programmer in Basic or any of the other popular programming languages needs to make is that between numbers as literal strings, and numbers as numerals.

When we write 2000 into a variable meant to contain the number of widgets on a warehouse shelf then it makes sense to treat 2000 as a numeric value. When the 2000 represents a postcode, it means nothing more than 'Sydney', and should be treated as a literal string.

Just when you think that you are getting the distinction clear, someone points out words like 'second' and 'ninth'. Are these terms more aligned with literals or with numbers?

Bill Atkinson, who wrote HyperCard, has dealt with this problem in a novel way. He treats every word or number as a literal – until the program attempts to undertake some computation. At this stage, HyperCard 'interprets' the string as a number and uses it in the calculation – returning the result to literal form again.

The point here is that you don't need to bother about making the distinction, since everything is a literal. Also, the interpreter can handle the word 'nine' in exactly the same way as the number '9' – or in fact, as the word 'ninth'. Up to 'ten' the program doesn't care. This is another of the reasons why HyperCard appears to use a very English-like language.

However there are limitations. A number can include only one period (.) representing the decimal point, and no other punctuation or spaces apart from a plus (+) or minus (-) sign at the front.

With calculations, you can have precision up to 19 decimal places, but this is unnecessary for most common uses. So you can set the precision you require by using the 'numberFormat' command –

```
set numberFormat to 0.00
```

This example would result in a string with at least one digit to the left of the period, and two to the right. If you don't want unnecessary zeros in the decimal places you can request –

```
set numberFormat to 0.####
```

and the trailing digits will only be added when they are non-zero, to a maximum of four.

HyperCard – potential and promise

Jeff Richardson, AMSEC consultant and computer education lecturer at the Gippsland Institute of Advanced Education, believes that HyperCard is the most significant innovation in personal computing since the release of the Macintosh itself.

YOU WILL probably have noticed that most articles about HyperCard deal with the question 'What is HyperCard?' 'What is so extraordinary about that?', I hear you ask. Well the very need to explain just what HyperCard is indicates that we have before us something very different.

We all share assumptions about what a computer is and the functioning of the the components that make up different system configurations. We share a notion of a 'word processor', a 'spreadsheet', a 'database' and other application and utility software; and we share a notion of a 'programming language' and an 'operating system.' We know, or at least think we know, what these things are, and when an innovation is made in any particular area there is no need to explain the basics of that area in itself. Not so with HyperCard. Sure, its a piece of software – but what is it?

Most of what I've read about HyperCard since its launch last August has attempted to answer this question. If you've been fortunate enough to use HyperCard yourself you'll probably understand why reviewers feel the need to stress so urgently that HyperCard is so much more than just a showy electronic filecard system.

I was initially very sceptical of HyperCard but after using it I became a total convert. I'm writing from the perspective of a HyperCard devotee, so be warned. HyperCard is the most significant innovation in personal computing since the release of the Macintosh itself.

HyperCard is a total computer environment: it's a modern and powerful programming language; it's a multipurpose application, able to manipulate and integrate text, graphics, data and complex calculation; it has a range of user entry levels that span from computer novice to application's programmer; it has the power to usurp the front end of the operating system, allowing the user to open any other application or utility from within their own customised HyperCard startup environment; and it can import other programs and run them, or use parts of them. It's simple enough for users to create their own applications without writing any program code at all. It can interface with the outside world through robotic control, digital sampling of sound and vision, through sound synthesis, and through video disk. It comes with a range of ready to use applications, but it is also open-ended, extensible and user definable. HyperCard empowers and emancipates the user in relation to software in the same way that the Macintosh does in relation to hardware.

At the heart of all this power is HyperCard's builtin programming language, HyperTalk. Like HyperCard itself, HyperTalk is not like other programming languages – and then again it is. But the languages it is descended from are not like the languages that are commonly used for program development. Programming in HyperTalk/HyperCard is a little bit like programming within a spreadsheet, attaching formulae to cells and sending output to other cells, but it is a lot more like programming in Logo and in Smalltalk.

Both these languages have been around for some time but have not been used by serious programmers who have recognised their revolutionary nature, because both languages have been held back by hardware which lacked the speed and memory to keep pace with their brilliance. HyperTalk is very much like Boxer, which is currently under development at the Massachusetts Institute of Technology – Boxer is a WYSIWYG system in a total sense; the entire system is visually represented and manipulable in a way that is similar to the Macintosh desktop, and the entire system, program, data, text, the lot, is always visible and available and alterable on the screen. Within Boxer, there is no distinction between an editor or any other part of the system. HyperCard/HyperTalk operates in the same spirit.

In HyperTalk it is messages, rather than commands which are the basis of getting things done. This is a subtle but profound difference from the imperative, sequential and procedural way in which most common computer languages are expressed. Messages are passed between objects and may use functions. User-defined objects and functions can be used in addition to those that are built in to HyperTalk. The language has a lexicon of primitives to meet the needs of the user. But it is very powerful and forgiving; it has a very high tolerance for natural language syntax and for the syntax of lower level computer languages. It is quite possible to write HyperTalk code in a pidgin of pseudo-code, debugging as you go, or as happens surprisingly often, HyperTalk gets your drift and is able to do what you want anyway. This style of programming is close to the ideal of 'descriptive programming', the most well known embodiment being the language Prolog, where the emphasis is on describing what is to be done rather than devising an algorithm to do it.

Modules of HyperTalk code, or scripts as they are called, are attached to objects. Objects are arranged in a hierarchy and this hierarchy determines how messages will be handled. Scripts are capable of altering other scripts and of altering themselves. There is no real distinction between programs and data. If all of this sounds confusing, I'm not surprised. HyperCard really does stretch and change the notion of just what programming is – try it and see for yourself!

Of the objects available in HyperCard, the button is the most immediate and dynamic. It responds to a click of the mouse, and this really lets you start pushing the computer around. You get the feeling that it will never kick sand in your face again. Yet for all this there are shortfalls. Other users will find their own gripes, but for me there are two types of object that are glaringly missing. They are sprites, the basis of any animation, and demons, the parallel processing tool for ordinary folk, lying silently in wait for a specified event. Both have been available in versions of Logo for well over five years now, and running on very dinky little machines.

Still, the whole HyperCard environment feels very Logo like. Certainly its the first Mac program that has made me seriously consider the machine as a realistic option for schoolchildren. And for me, HyperCard is the best realisation so far of the potential and promise I felt when I first sat down in front of a Macintosh. □

Range is established by the word 'to' –

```
repeat with increment = 1 to 19
```

Along with these literals and numbers you will need to use operators such as equalTo (=), greaterThan (>), lessThan (<), and so on, to write command lines like this one –

```
if usersAge > 45 and usersAge < 60 put "Middle Aged" into card
field "ageClass"
```

As in most languages, parentheses are used for groupings, with the innermost expression being evaluated first. There are the normal +, -, *, and / arithmetical operators, and also the logical operators =, >, <, and ≷, and, or, and not, as well as a few other more obscure ones. You can use either the words 'is not' or the symbols ≷ to mean 'not equal to', and the words 'contains', 'is in' and 'is not in' are all evaluated to be either true or false.

The words 'true' and 'false' are both constants recognised by the program along with 'up', 'down', 'left', 'right', 'space' (for example: " "), 'tab', 'return', 'empty' (for example: the null character ""), 'return', 'lineFeed', 'formFeed', and 'pi'.

And, to end this brief four-part look at HyperTalk, a quick tip on security. If you want to ensure that no one can modify one of your stacks try this handler at the Stack level –

```
on openStack
   set userLevel to 1
end openStack
```

This allows the user to Browse only (userLevel 2 allows them to type). There are ways around this script through the message box, of course, but it's a simple handler which makes access to change difficult for the less experienced.

I was going to conclude this part with three substantial scripts that I developed for importing text, spreadsheet and database files into HyperCard. But Version 1.2 landed on my desk last week and its utilities disk includes ready-made handlers for both importing and exporting files – and I have to admit that they are better than the ones I wrote.

So if you are still staggering along with the early version, take your current HyperCard disk down to the nearest Mac dealer and demand the upgrade! □

User-defined Functions
for structured programming in Basic

Is your code impossible to read? Cluttered? Hard to maintain? A hassle to move to a different version of Basic? — Then you need User-Defined Functions from Jeff Richards!

USER-DEFINED functions are one of the few aids to structured programming available to the Basic programmer. Carefully using them can lead to programs that are easier to understand and maintain. In many cases the program will also be more compact and will execute faster.

The examples presented here are in Microsoft Basic, and should work for most of its implementations. They will also apply to other Basics that support user-defined functions, with the possible exception of using logical expressions as values.

The definition of a user-defined function is —

```
DEF FNname[(parameter0__list)]
=expression
```

and the function is used as if it was an expression, in the form —

```
FNname(argument0__list)
```

For example, the definition of a simple mathematical function to calculate the distance between two points on a plane (using cartesian coordinates), is —

```
10 DEF FNDIST(X1,Y1,X2,Y2)
   =SQR((X2—X1)↑2+(Y2—Y1)↑2)
```

A typical reference, which gives the distance from the origin to the point (A,B), might be —

```
100 PRINT FNDIST(0,0,A,B)
```

In order to evaluate the function the value 0 is substituted for X1 and Y1, the value of A is substituted for X2 and the value of B for Y2. The statement is functionally identical to —

```
100 PRINT SQR((A—0)↑2+(B—0)↑2)
```

This demonstrates why the variables named in the parameter list are referred to as dummy variables. When the function is evaluated, these variables are replaced by the actual values (constants or variables) taken from the argument list. Thus it is important that the values in the argument list match, in type and number, the dummy variables specified in the parameter list. It is important to remember that any variables in the program that may share the same name as variables in the parameter list (X1,Y1,X2 and Y2 in this example) are not affected by the use of the function.

If a variable in the expression part of the function does not appear in the parameter list, then the current value of the variable is used, or 0 if the variable has not yet been assigned a value. It is good programming practice to include all variables required for a function in the parameter list (and some would claim that Basic should enforce this rule). A possible exception would be 'global' variables such as pi = 3.14159.

It should also be noted that functions have a 'type' — integer, single precision, double precision or string — in exactly the same way that variables do. The type of a function is determined by the special character appended to the function name, the effect of a 'DEF type' statement, or the default type.

Mathematical functions are a typical use for user-defined functions. Microsoft Basic requires that the arguments to the transcendental functions — SIN(), COS() and TAN() — are in radians. For many problems it is convenient if the argument is in degrees. This conversion can be implemented in user-defined functions —

```
DEF FNSINE(N)=SIN(N/57.2958)
```

Similarly, the unimplemented functions SEC(), COSEC() and COT(), can be easily implemented as user-defined functions using the supplied intrinsic functions.

The are many reasons for using a function, but the best one is *simplicity*. Any expression that is reasonably complex and is used repeatedly, is a candidate for a function. This will make the code less cluttered, easier to read, more maintainable, and easier to move to a different version of Basic. An example of a function that simplifies coding, and also demonstrates a function of type string, is —

```
20 DEF FNINS$(A$,B$,N)=
   LEFT$(A$,N)+B$+RIGHT$
   (A$,LEN(A$)—N)
```

This function inserts the string B$ after position N in string A$. It might be used in the simple expression —

```
200 C$=FNINS$("abcdefgh","f",5)
```

— which would assign the string "abcdefgh" to C$. The advantages in elimination of repetitive code, and the possibility of coding errors, should be a obvious.

Other procedures that are candidates for functions are not so obvious. Because functions must evaluate an expression to return a single result, they cannot be used for routines that require an IF statement or a FOR-NEXT loop. However, many routines can be forced into a function format by using logical expressions. A logical expression is a logical relation that is used to return a value — in Microsoft Basic the value is –1 for true and 0 for false. Thus the expression A=(B=C) would set A to –1 if B was equal to C, or it would set A to 0 if B was not equal to C.

A typical use for this feature is in a function to find the maximum (or minimum) of two values. This would usually be expressed as IF B>C THEN A=B ELSE A=C

The are many reasons for using a function, but the best one is simplicity. Any expression that is reasonably complex and is used repeatedly is a candidate for a function.

However, as a single expression it can be recast as —

```
30 DEF FNMAX(A,B)=
   -A*(A>B)—B*(B=>A)
```

In this case, if A is greater than B, the expression becomes –A*(—1)—B*(0), which reduces to A. If A is less than or equal to B then it becomes –A*(0)-B*(-1) which reduces to B. There are other ways of achieving the same result, but this is as convenient as any.

Similar use of logical expressions can simplify otherwise tedious tasks. A common requirement when dealing with operator input is to convert lower case characters to upper case before testing. A function that uses logical expressions can do the job simply —

```
40 DEF FNUPPER$(A$)=
   CHR$(ASC(A$)+32*(ASC(A$)>96
   AND ASC(A$)<123))
```

This can be translated as 'Take the ASCII value'of the character. If it lies between 96 and 123 then subtract 32. Convert the number back to a character and return it as the result of the function.'

Note the use of the intrinsic functions ASC() and CHR$() in this example. In fact, any function, intrinsic or user-defined, can be referenced in a user-defined function. This makes 'stacking' of functions possible as the next example shows. This function uses FNUPPER$ to convert the first character of a string to upper case —

```
50 DEF FNCAPS$(A$)
   =FNUPPER$(A$)+RIGHT$
   (A$,LEN(A$)—1)
```

String functions are particularly useful in user-defined functions. Allied with the need to convert operator input to upper case, is the need to check the input for validity. A simple function that compared a character with a predefined set of alternatives might look like this —

```
60 DEF FNCHOICE(A$)=
   INSTR("ABCDEF",FNUPPER$(A$))
```

Such a function returns 0 if the character was invalid, or a number between 1 and 6 representing the characters 'A' to 'F' in upper or lower case. It would be used in a typical program with —

```
600 INPUT ANS$:IF FNCHOICE(ANS$)=
    0 GOTO 600
```

But any intrinsic function can be used in a user-defined function, including some that may not appear to be simple functions. For instance, INPUT$() is a function. It could be used in the above example to get the input from the operator. This also simplifies the use of the function.

```
70 DEF FNSELECT=
   INSTR("ABCDEF",FNUPPER$
   (INPUT$(1)))
700 WHILE NOT FNSELECT:WEND
```

In practice, it would be preferable to make the function more general and to indicate in the use of the function just what was going on —

```
70 DEF FNSELECT(A$)=
   INSTR(A$,FNUPPER$(INPUT$(1)))
700 WHILE NOT FNSELECT
   ("ABCDEF"):WEND
```

Such a function could also be used directly to control program flow —

```
700 ON FNSELECT("ABCDEF")
   +1 GOTO 700,800,..
```

Other tricks are available to permit procedures to be coded into functions rather than as subroutines or IF statements. A requirement to return the description associated with a number, for instance the month of the year, given a date, might be tackled with a string array initialized to the month names and referenced on the month number.

Such a procedure requires that a variable name be set aside for the array, and several lines of initialization code will be required at the start of the program. Using a function to perform the job eliminates the use of a special variable, and the initialization is reduced to the single DEF FNname statement.

The problem is easy if the strings are all the same length, but can still be worthwhile even when the strings differ in size. An example that displays the status of a series of points in a network might be —

```
80 DEF FNSTATE$(N)=
   MID$(" ONOFF ?",N*3—2,3)
800 FOR I=1 TO MAXSW:PRINT I,
   FNSTATE$(ST(I)):NEXT
```

Note that the strings — "ON", "OFF" and "?" — have been forced to the same length to simplify things.

There are circumstances when functions can be used to implement a 'brute force' programming technique that would be unmanageable if it had to be repeatedly coded in-line, but is much faster than a subroutine call. An example is the problem of counting the number of '1' bits in a character — a problem common in error checking procedures. A function to do the counting is —

```
90 DEF FNCOUNT(A)=
   (A AND 64)/64+(A AND 32)/32+
   (A AND 16)/16+(A AND 8)/8+
   (A AND 4)/4+(A AND 2)/2+(A AND 1)
```

This function operates about twice as fast as a subroutine and a FOR-NEXT loop. A matching function to return the character with its high-order bit set, if required, to ensure an even number of '1' bits is —

```
91 DEF FNPEVEN$(A$)=
   CHR$(ASC(A$)OR &H80*
   (FNCOUNT(ASC(A$))MOD 2))
```

And a final example, given in Listing 1, demonstrates the value of functions in a library of 'tools' — procedures that perform defined tasks and can be included in any program merely by copying from a library of procedures. In this usage the complexity of the function is irrelevant. What might be considered overly complex and opaque coding is acceptable in a tool that has been carefully debugged, and documented. □

```
99 DEF FNDAY$(D,M,Y)=MID$('SunMonTueWedThuFriSat',((D+2*
      (3-(Y\100 MOD 4))+(Y MOD 100)\12+(Y MOD 100 MOD 12)
      +(Y MOD 100 MOD 12)\4+VAL(MID$('033684625032',M,1))
      -10*(M=12)+((M<3) AND (Y/4=Y\4) AND NOT
      (Y/400<>Y\400)))MOD 7)*3+1,3)
```

Listing 1. *The function returns the day of the week for any date in the Gregorian calendar after September 13, 1752 (when the calendar was adjusted by four days). It uses most of the tricks described, including logical expressions. It also uses substrings referenced on a variable for both the day name and the magic number for the month (but note that the magic number 12 for December needs special handling). Integer divide ('x\y') is a briefer way of expressing* INT(x/y).

Byting ProDOS Back - Part 1

Stewart Fist offers Apple users a basic guide to how ProDOS handles files – with special reference to keeping Appleworks files on a hard disk.

BY NOW, most Apple users are familiar with ProDOS, and despite initial reservations, most of us have learned to accept it as a better operating system than the old DOS 3.3.

At the time Apple changed over to ProDOS, it wasn't overly apparent to many people why the swap was necessary. DOS 3.3 was one of the great "user-friendly" operating systems in the evolutionary path of computers, and ProDOS will never ignite the same affection in users. ProDOS was certainly "bigger, faster and more powerful" but it has the chrome-and-plastic slickness of a modern Japanese car, as opposed to the rugged reliability of a faithful old FJ Holden.

But ProDOS is here to stay. We've had to face the fact that even small-system computers like the Apple II are fast moving into an era of massive mass-storage systems. Not just our current magnetic hard disk systems (though the newly developed R-DAT recording techniques seem to indicate that magnetic technology is still hot in the running), but also CD-ROM and the exotic optical storage systems of the future.

ProDOS can handle up to 32 megabytes in one volume and 16 megabytes in any one file, so it'll do most Apple users for the time being.

When computer users face the prospects of having hundreds, or thousands of files on a single mass-storage device, they run up against the limitations of the human brain. A computer can flick through a thousand-entry catalog in a fraction of a second, but a human can't. So the cataloging system must be designed in a branched 'hierarchical' structure if we are ever going to find anything on a large disk.

ProDOS's directory structure therefore makes sense for anything with more capacity than the standard floppy drives — although it must be admitted that the use of multi-level pathnames is annoying for single-sided floppy disk users, on occasions. And when you're trying to find, repair or recover information on a disk, a branched directory structure adds to the complexity.

Saving Space with Blocks

When a ProDOS disk is first formatted, only one 'block' of disk space is reserved for the main 'volume' directory. This volume directory can carry a maximum of 51 file entries, but each of these entries can itself be a subdirectory occupying another block of disk space — and that block can be anywhere at all on the disk.

This means that the operating system doesn't need to allocate a large number of disk blocks for directory use initially — it does this progressively when needed. The volume directory and the subdirectories are simply treated by ProDOS as different forms of a normal file, so you don't get space allocated for, but not used by, subdirectories.

What ProDOS gives, ProDOS also takes away. While the directory allocation system saves wasted disk space on floppies,

where subdirectories aren't usually needed, ProDOS often does waste disk space through the use of 'blocks' rather than 'sectors' as the primary disk division (see the Track/Sector to Block Conversion Table for the conversion calculations).

By definition, there are two disk sectors to each block, so sectors are always allocated in pairs. Normally, this would mean a textfile with only a couple of words would still take up two sectors of disk space (one block), in addition to the directory and formatting information.

To avoid this 'space waste', ProDOS treats these short files in a different way to the longer ones. For instance, Appleworks files with under 200 characters (about 30 words) are registered with the operating system as 'seedling' files and treated quite differently to the longer 'sapling' files. (Non-Appleworks 'seedling' files can be up to 256 characters in length — which still isn't much.)

ProDOS manuals and articles talk a lot about 'seedling', 'sapling' and 'tree' files, without ever getting down to the nitty-gritty of what these terms mean in a practical sense. To understand this you've got to look at the way ProDOS stores information on the disk itself.

Inspect Your Disk

So if you want to understand ProDOS you should first invest in a good disk-inspection program. Pro-Byter, from Beagle Brothers (at US$39.95), is probably the best value for money at the present time. Send a Visa card or Mastercard number to Beagle Brothers, at 3990 Old Town Avenue, San Diego CA 92110, and ask for it to be sent airmail — or you'll be waiting three months.

If you've got either the old DOS 3.3 version of 'Apple Mechanic' or the 'Tricky Dick' disk-zapping program, you can use them to look at a ProDOS floppy, but initially you'll find the sector allocation quite confusing. This is complicated by the fact that two sectors are equal to one block (usually designated Part A and Part B), and these sectors aren't always alongside each other, as can be seen in the Conversion Table.

Looking at the ProDOS disk itself, the main (volume) directory now starts at Block $02 Part A (all block numbers are in hex), which in the old terminology would be Track $00 Sector $0B. If your disk holds more than five files, the second part of this directory (Part B) would be found at Track $00 Sector $0A.

Another part of the disk which you should identify is Block $06 (Track $00, Sector $03/$02), which is reserved for the bit-map. This bit-map is checked by ProDOS whenever it is writing to a disk to see which blocks are free for use; it works like a large sheet of graph paper on which the computer notes whenever a block is used. We'll go into how this works later in the series.

Operation Rescue

ProDOS has a simple system of allocating disk space. It uses the lowest block number available and then moves to the next block. This makes it fairly easy to track a file through a disk if you've accidentally deleted it; you always start from the lowest block number and then move progressively up. You might not find that every free block you come across contains some of your files (some might have been used before for another file), but you can be quite sure that your file will be in numeric sequence.

If you've both deleted and overwritten a long textfile with a much shorter one, the chances are that a significant part of the text will still be recoverable. The overwrite

Sector Offset Chart

Sector No.	Offset + Part	Sector No.	Offset + Part
$00	0 A	$08	3 B
$01	7 A	$09	3 A
$02	6 B	$0A	2 B
$03	6 A	$0B	2 A
$04	5 B	$0C	1 B
$05	5 A	$0D	1 A
$06	4 B	$0E	0 B
$07	4 A	$0F	7 B

Track/sector to Block Conversion

TRACK	$00	$01	$02	$03	$04	$05	$06
	Hex Dec	Dec	Dec				
$00	$00 A 0 A	7 A	6 B	6 A	5 B	5 A	4 B
$01	$08 A 8 A	15 A	14 B	14 A	13 B	13 A	12 B
$02	$10 A 16 A	23 A	22 B	22 A	21 B	21 A	20 B
$03	$18 A 24 A	31 A	30 B	30 A	29 B	29 A	28 B
$04	$20 A 32 A	39 A	38 B	38 A	37 B	37 A	36 B
$05	$28 A 40 A	47 A	46 B	46 A	45 B	45 A	44 B
$06	$30 A 48 A	55 A	54 B	54 A	53 B	53 A	52 B
$07	$38 A 56 A	63 A	62 B	62 A	61 B	61 A	60 B
$08	$40 A 64 A	71 A	70 B	70 A	69 B	69 A	68 B
$09	$48 A 72 A	79 A	78 B	78 A	77 B	77 A	76 B
$0A	$50 A 80 A	87 A	86 B	86 A	85 B	85 A	84 B
$0B	$58 A 88 A	95 A	94 B	94 A	93 B	93 A	92 B
$0C	$60 A 96 A	10. A	102 B	102 A	101 B	101 A	100 B
$0D	$68 A 104 A	111 A	110 B	110 A	109 B	109 A	108 B
$0E	$70 A 112 A	119 A	118 B	118 A	117 B	117 A	116 B
$0F	$78 A 120 A	127 A	126 B	126 A	125 B	125 A	124 B
$10	$80 A 128 A	135 A	134 B	134 A	133 B	133 A	132 B
$11	$88 A 135 A	143 A	142 B	142 A	141 B	141 A	140 B
$12	$90 A 144 A	151 A	150 B	150 A	149 B	149 A	148 B
$13	$98 A 152 A	159 A	158 B	158 A	157 B	157 A	156 B
$14	$A0 A 160 A	167 A	166 B	166 A	165 B	165 A	164 B
$15	$A8 A 168 A	175 A	174 B	174 A	173 B	173 A	172 B
$16	$B0 A 176 A	183 A	182 B	182 A	181 B	181 A	180 B
$17	$B8 A 184 A	191 A	190 B	190 A	189 B	189 A	188 B
$18	$C0 A 192 A	199 A	198 B	198 A	197 B	197 A	196 B
$19	$C8 A 200 A	207 A	206 B	206 A	205 B	205 A	204 B
$1A	$D0 A 208 A	215 A	214 B	214 A	213 B	213 A	212 B
$1B	$D8 A 216 A	223 A	222 B	222 A	221 B	221 A	220 B
$1C	$E0 A 224 A	231 A	230 B	230 A	229 B	229 A	228 B
$1D	$E8 A 232 A	239 A	238 B	238 A	237 B	237 A	236 B
$1E	$F0 A 240 A	247 A	246 B	246 A	245 B	245 A	244 B
$1F	$F8 A 248 A	255 A	254 B	254 A	253 B	253 A	252 B
$20	$0100 A 256 A	263 A	262 B	262 A	261 B	261 A	260 B
$21	$0108 A 264 A	271 A	270 B	270 A	269 B	269 A	268 B
$22	$0110 A 272 A	279 A	278 B	278 A	277 B	277 A	276 B

would have occurred firstly in those lower block numbers that contained the index pointers (before the text starts), so you might only have lost the first few hundred characters.

In fact if the overwrite file was less than 200 characters (a 'seedling' file) in length, you probably won't have lost any of the text at all and full recovery will be possible, if not easy. This will become more understandable when we look at the different formats taken on by files on the disk, in subsequent articles.

The Volume Directory Block

For starters, let's look at the volume directory block and try to make sense out of the jumble. The format you see in Figure 1 is that produced by Pro-Byter; it has some of the bytes translated to hex figures, and others to ASCII characters. Other disk-inspection programs might look quite different on the screen, but the information contained will be the same — and in the same sequence, since this is the order in which the directory bytes have been read off the disk.

At the top of the screen shown in Figure 1 is a single line containing two sets of dual hex pairs. The first dual pair (A) tells you where to find the directory's previous block ($0000 since there isn't a previous block), while the second (B) tells you where to look for the following block.

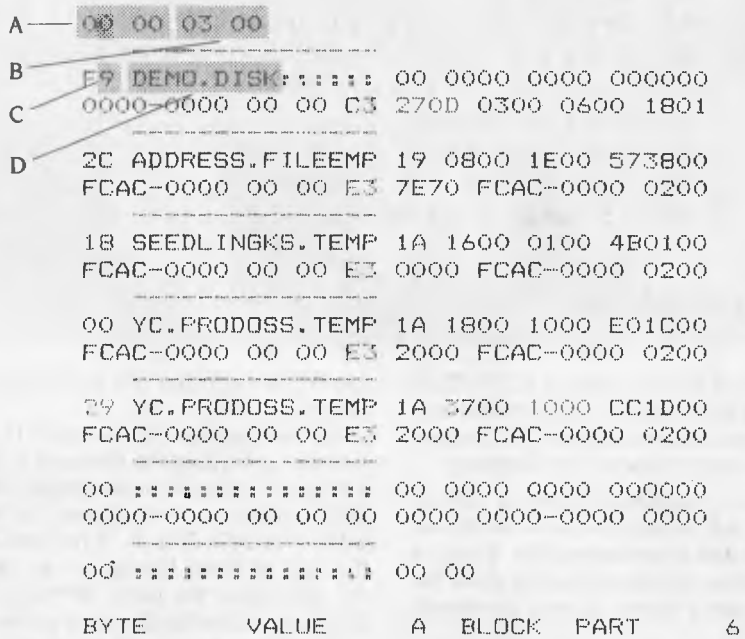
This main volume directory is always at Block $02, so Block $0003 has been reserved for future use. Remember you always need to read dual hex pairs 'backwards' — the least significant pair ($03) comes first, and the most significant ($00) comes last.

The first entry in the directory file is the volume name (C), which in this case is DEMO.DISK. We know it is a volume name from the hex number that precedes it (F9).

If you look at Figure 2, you will see that the first part of this hex pair indicates the type of file storage on the disk — in this case, it's a Type 'F' file storage which indicates a volume directory. ADDRESS.FILE below is a storage type '2', which indicates a sapling file of over 256 bytes but less than 128,000(or 257 blocks). Most of your word processing files will be of this type.

The file named SEEDLING is, as can be assumed from the name, a storage type '1'. Its full contents is the sentence "I am a seedling file."

Figure 1. *The volume directory block as seen by Pro-Byter. A is the location of the previous block; B is the location of the following block; C is the volume name; and D is the filename length.*



$07	$08	$09	$0A	$0B	$0C	$0D	$0E	$0F
4 A	3 B	3 A	2 B	2 A	1 B	1 A	0 B	7 B
12 A	11 B	11 A	10 B	10 A	9 B	9 A	8 B	15 B
20 A	19 B	19 A	18 B	18 A	17 B	17 A	16 B	23 B
28 A	27 B	27 A	26 B	26 A	25 B	25 A	24 B	31 B
26 A	35 B	35 A	34 B	34 A	33 B	33 A	32 B	39 B
44 A	43 B	43 A	42 B	42 A	41 B	41 A	40 B	47 B
52 A	51 B	51 A	50 B	50 A	49 B	49 A	48 B	55 B
60 A	59 B	59 A	58 B	58 A	57 B	57 A	56 B	63 B
68 A	67 B	67 A	66 B	66 A	65 B	65 A	64 B	71 B
76 A	75 B	75 A	74 B	74 A	73 B	73 A	72 B	79 B
84 A	83 B	83 A	82 B	82 A	81 B	81 A	80 B	87 B
92 A	91 B	91 A	90 B	90 A	89 B	89 A	88 B	95 B
100 A	99 B	99 A	98 B	98 A	97 B	97 A	96 B	103 B
108 A	107 B	107 A	106 B	106 A	105 B	105 A	104 B	111 B
116 A	115 B	115 A	114 B	114 A	113 B	113 A	112 B	119 B
124 A	123 B	123 A	122 B	122 A	121 B	121 A	120 B	127 B
132 A	131 B	131 A	130 B	130 A	129 B	129 A	128 B	135 B
140 A	139 B	139 A	138 B	138 A	137 B	137 A	136 B	143 B
148 A	147 B	147 A	146 B	146 A	145 B	145 A	144 B	151 B
156 A	155 B	155 A	154 B	154 A	153 B	153 A	152 B	159 B
164 A	163 B	163 A	162 B	162 A	161 B	161 A	160 B	167 B
172 A	171 B	171 A	170 B	170 A	169 B	169 A	168 B	175 B
180 A	179 B	179 A	178 B	178 A	177 B	177 A	176 B	183 B
188 A	187 B	187 A	186 B	186 A	185 B	185 A	184 B	191 B
196 A	195 B	195 A	194 B	194 A	193 B	193 A	192 B	199 B
204 A	203 B	203 A	202 B	202 A	201 B	201 A	200 B	207 B
212 A	211 B	211 A	210 B	210 A	209 B	209 A	208 B	215 B
220 A	219 B	219 A	218 B	218 A	217 B	217 A	216 B	223 B
228 A	227 B	227 A	226 B	226 A	225 B	225 A	224 B	231 B
236 A	235 B	235 A	234 B	234 A	233 B	233 A	232 B	239 B
244 A	243 B	243 A	242 B	242 A	241 B	241 A	240 B	247 B
252 A	251 B	251 A	250 B	250 A	249 B	249 A	248 B	255 B
260 A	259 B	259 A	258 B	258 A	257 B	257 A	256 B	263 B
268 A	267 B	267 A	266 B	266 A	265 B	265 A	264 B	271 B
276 A	275 B	275 A	274 B	274 A	273 B	273 A	272 B	279 B

```
$0 Inactive file
$1 Seedling file (1 block)
$2 Sapling file (max 257 blocks)
$3 Tree file (over 257 blocks)
$4 Pascal area
$D Subdirectory file entry
$E Subdirectory header
$F Volume directory
```

Figure 2. *File storage types.*

The first of the files named YC.PRODOS shows a storage type '0'. This indicates it has become inactive because it's been re-saved (directly below in the directory).

Whenever ProDOS re-saves a file already on disk, it first creates a completely new copy, and then deletes the reference to the old file. It does not save the new file into the same block as was previously used.

Most operating systems work this way, but it is often not recognised that if a mistake is made, and you accidentally save some incorrect data under an old filename, the original data can still be recovered intact, by using Pro-Byter to make some subtle changes to a few bytes on the disk. It's not just a simple matter of changing the storage type $00 to $29, but we'll get to this later.

The second byte (D in Figure 1) in the hex pair preceding the filename is simply a measure of the filename length. DEMO.DISK measures 9 characters in length, while ADDRESS.FILE is 12 (in hex $C). On the end of these filenames can be seen the residual of the name APPLEWORKS.TEMP added to the directory by the operating system as a preliminary to saving the file. ProDOS doesn't wipe out the old filenames, it just overlays them and records how many characters are in use with this second byte of the hex pair.

Next month we'll look at the other types of information contained in the ProDOS directory, and find out how to change them. □

HEX NUMBERS

a) Each digit in a hex pair (from 0 to 9, then A to F) represents a decimal number from 0 to 15.

b) A hex pair can therefore represent any decimal number from 0 ($00) to 255 ($FF).

c) Dual hex pairs can represent any decimal number from 0 ($0000) to 65,535 ($FFFF).

d) From a disk or memory, dual pairs are read 'backwards'. The least significant number comes first — so $04 followed by $01, translates to $0104, which is decimal 260.

e) Remember when you are counting with hex numbers that zero comes first. If a pointer is indicating $01, it is pointing at the second byte in the series, not the first.

Byting ProDOS Back - Part 2

How can a seedling take up a whole block?

BEFORE we move on, let's review the main points in Part 1 – ProDOS allocates disk space in blocks rather than sectors (there are two blocks per sector). Free space is allocated in a strict sequence from the lower block number to the higher and, for this reason, ProDOS itself will always occupy the lowest blocks on a standard floppy. Block six is the bit-map, recording which blocks on the disk are in use and the primary directory is at block two, with subdirectories allocated as needed and treated as normal files.

Files are treated in three different ways, according to length. If the files are less than 512 characters long (212 for Appleworks files) they are called 'seedling' files and are stored in only one block. If they are medium length (up to 128 Kbytes) they are called 'saplings', and are stored in from three to 256 blocks. Larger files are called 'trees'.

When you re-save a file (supposedly writing over one already on the disk), ProDOS firstly writes the whole new file into spare space, and then changes the file header and the bit map to indicate the superseded file is no longer in use.

The bytes preceding the volume (disk) name in the directory block (see Figure 1) indicate: A the hex number of the previous block — $0000 in this case; and B the block number of space allocated for main directory extension — $0003 (read the bytes in reverse order, remember!)

The byte preceding each file name uses two half bytes to give us C the storage type, and D the length of the file name. These two half bytes are changed to zero when a file is inactive.

Deeper into the Directory Block

The bytes following the file name in the directory provide a lot of information about how and where data is stored on the disk. Fortunately, Pro-Byter — which printed out the directory in Figure 1 — groups these bytes together to make them more comprehensible.

You'll see in Figure 1, directly to the right of the file name E a single byte F which tells us the file type. ProDOS has made provision for a large number of file types; you can see the major ones in Figure 2.

File type $19 (the file type of ADDRESS.FILE) is an Appleworks database program, while the SEEDLING file below is from the word-processing section of Appleworks. Apple has made provision for a whole range of file types, from Pascal to Binary

and Integer BASIC, and there are even some types reserved for user definition.

The double byte following this — G — points to the file's first block. From this it can be seen ADDRESS.FILE starts at block $0008 (reverse the byte order) on the disk.

BOFs and EOFs

This first-block pointer remains in the directory listing even though the file header shows the file is inactive. You will see this with the first of the YC.PRODOS directory entries below; it points to block $0018 (note the YC.PRODOS is an Appleworks word-processing-type file, not a system file).

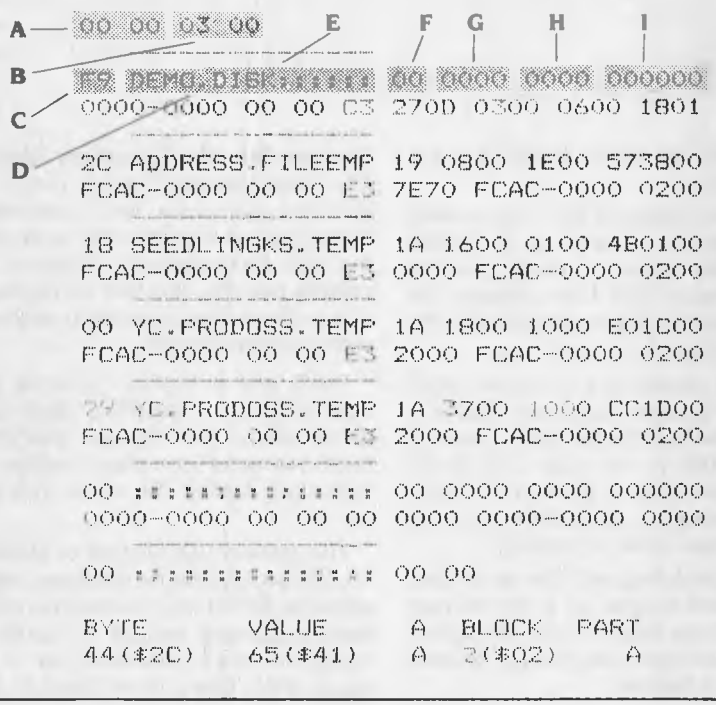
This means you can find the first block (which contains a directory of the other blocks) of a file that has been deleted — whether it has been deleted through deliberate action, or by re-saving the file to disk, as in this example.

The size of the file in terms of the number of blocks allocated for its use is shown at H. The ADDRESS.FILE takes up $001E blocks (that's 30 blocks in decimal), while the seedling file below only takes up one block. If the word-processing document renamed SEEDLING was to be expanded and re-saved to disk, the number of blocks consumed would immediately jump to at least three, since the file would then become a 'Sapling'. The new file header would change from $18 to $28, to indicate this change in status.

File Type	Preferred Use
$00	Typeless file
$01	Bad block file
$04	ASCII text file (SOS and ProDOS)
$06	General binary ile
$08	Graphics screen file
$0F	Directory file (SOS and ProDOS)
$19	AppleWorks Data Base file
$1A	AppleWorks Word Proc. file
$1B	AppleWorks Spreadsheet file
$EF	Pascal area
$F0	ProDOS CI added command file
$F1-$F8	ProDOS user defined files 1-8
$F9	ProDOS reserved
$FA	Integer BASIC program file
$FB	Integer BASIC variable file
$FC	Applesoft program file
$FD	Applesoft variables file
$FE	Relocatable code file (EDASM)
$FF	ProDOS system file

Figure 2. *File types under* ProDOS.

Figure 1. *The volume directory block as seen by* Pro-Byter. **A** *is the hex number of the previous block;* **B** *is the block number of space allocated for the main directory extension;* **C** *shows the file type;* **D** *indicates the length of the file name;* **E** *is the file name;* **F** *tells the file type;* **G** *points to the file's first block;* **H** *is the space allocated to the file; and* **I** *is the actual lenght of the file.*



The next group of three bytes I gives us information about the actual length of the file (as distinct from the number of blocks allocated to it, which includes index blocks in larger files). This information is useful for finding the end of the file.

This next bit is a little complex, but very important, so read carefully. Reading the hex bytes from the right and using the ADDRESS.FILE example again, this group of three bytes tells us the file is $0038 sectors (note, 'sectors' or half blocks — not blocks) long, plus $57 bytes. Counting starts from the first sector with formatting information. This gets a bit confusing because it doesn't always relate directly to the number of blocks allocated. Remember when you are using hex numbers you must always count from zero. Thus, $00 is the first number in the sequence.

For instance, a seedling file can only occupy one block, and so Appleworks uses the first half block (Part A) for format information (tab settings, line-break information and so on) and the second half block (Part B) to actually store the text. The larger sapling or tree files reserve blocks for indexes as well.

The seedling file named SEEDLING therefore shows the end-of-file (EOF) marker is set at sector $0001 (that is, the second sector), and will be found at the $4Bth (75th) byte. Figure 3 shows the Pro-Byter printout of the block that makes up this seedling file. Pro-Byter gives us an ASCII translation in the left-hand column, and the actual bytes in groups of four, in the three columns on the right.

```
Part A - Formatting Information.

0":::0=====|=   30220000 4F3D3D3D 3D3D7C3D
===|====|===   3D3D3D7C 3D3D3D3D 7C3D3D3D
=|====|====|   3D7C3D3D 3D3D7C3D 3D3D3D7C
====|====|==   3D3D3D3D 7C3D3D3D 3D7C3D3D
==|====|====   3D3D7C3D 3D3D3D7C 3D3D3D3D
|====|====|=   7C3D3D3D 3D7C3D3D 3D3D7C3D
===|====|===   3D3D3D7C 3D3D3D3D 7C3D3D3D
::,":::::::::   00002C22 0300000A 00000000
::::::::::::   00000000 00000000 00000000
::::::::::::   00000000 00000000 00000000
::::::::::::   00000000 00000000 00000000
::::::::::::   00000000 00000000 00000000
::::::::::::   00000000 00000000 00000000
::::::::::::   00000000 00000000 00000000
::::::::::::   00000000 00000000 00000000
::::::::::::   00000000 00000000 00000000
::::::::::::   00000000 00000000 00000000
::::::::::::   00000000 00000000 00000000
::::::::::::   00000000 00000000 00000000
::::::::::::   00000000 00000000 00000000
::::           00000000 TRACE:SEEDLING
BYTE      VALUE      =  BLOCK  1 OF 1
83($53)   61(S3D)    =  22($16)   A     2

Part B - Text.

::::::::::::   00000000 00000000 00000000
::::::::::::   00000000 00000000 00000000
::::::::::::   00000000 00000000 00000000
::::::::::::W  00000000 00000000 19000097
This is a se   54686973 20697320 61207365
edling file:   65646C69 6E672066 696C6500
P  ::::::::::  D0FFFF00 00000000 00000000
::::::::::::   00000000 00000000 00000000
::::::::::::   00000000 00000000 00000000
::::::::::::   00000000 00000000 00000000
::::::::::::   00000000 00000000 00000000
::::::::::::   00000000 00000000 00000000
::::::::::::   00000000 00000000 00000000
::::::::::::   00000000 00000000 00000000
::::::::::::   00000000 00000000 00000000
::::::::::::   00000000 00000000 00000000
::::::::::::   00000000 00000000 00000000
::::::::::::   00000000 00000000 00000000
::::::::::::   00000000 00000000 00000000
::::           00000000 TRACE:SEEDLING
BYTE      VALUE     c@  BLOCK  1 OF 1
263($107) 0($00)    @   22($16)   B     2
```

Figure 3. *Printout of* SEEDLING *file.*

The bottom lines of the screen provide information on the file name, the cursor byte and block numbers, and whether this is Part A or B. The value category provides a translation of the byte under the cursor.

You can see in Figure 3, Part A is reserved for the formatting information. The ASCII translation shows a series of '|====' characters which are actually the default TAB settings.

I haven't been able to work out fully what all these format bytes represent, and the Apple documentation I have is obviously incomplete. It appears some of the bytes just contain random numbers with no meaning, so I'd be interested to hear from anyone with more information.

Figure 4 shows what the bytes in Figure 3 represent, reading down from the top of Part A (remember, this is an Appleworks word-processing file we're talking about).

Byte	Description
0 - 3	Not used
4	Always $4F for some reason!
5 - 84	Tab stops
85	The zoom switch (boolean)
86 - 89	Not used
90	Are page-break lines showing (boolean)?
91	The minimum left margin (figured in tenths of an inch)
92 - 249	Reserved for later mail-merge facility
250 - 299	Left available for anyone to use.

Figure 4. *Explanation of* SEEDLING *file listing.*

At the bottom of Figure 4, we have started to describe the first couple of lines of Part B, with the file proper starting from the 300th character (although Appleworks calculates everything itself from the first character of the formatting bytes).

So although the example named SEEDLING only contains the words "I am a seedling file" it still occupies the best part of the whole block. The three-byte file-length group I in the directory tells us the end of the file will be on the second sector ($0001), and counting from the first byte on this sector, we will find the first unused byte at position $4B, which is the 75th byte (count the leading zero).

Okay, time for some maths:
1 sector = 256 characters + 75 = 331
then subtract 300. This suggests the text-file length is 31 characters, when in fact it is only the words "This is a seedling file" — 23 characters. Where do the other eight bytes come from?

Two bytes come from the $FFFF end-of-file marker (EOF), and there are two bytes ($001D0) which mark the last carriage return. Up front, there is an extra double byte followed by two single bytes in sequence — $0019 $00 $97 ($19000097) in the example. But before we look at these, let's clear up the problem of the end-of-file marker.

In Figure 3 you can see the ASCII translation has added a 'P' to the end. The 'P' is a high-byte translation of $D0, which is the carriage-return byte. In this case, it precedes the $FFs at the end of the file, but you'll often find them scattered through the text.

Immediately before this $D0 (carriage return) at the end of the file, is a byte $00, which is used to indicate the horizontal screen position of the carriage return. It can obviously be any number below $50 (80) for the normal 80-column screen.

Formatting Commands

Any byte higher than $D0 (that is, from $D1 to $FF) is a formatting command. If the SEEDLING file were longer you would see some of these scattered through the text bytes, but since this file is only one sentence long, it doesn't need any. In the next part of this series I'll give you the main formatting translations for Appleworks word-processing files.

Now, let's return to the front-of-file sequence $0019 $00 $97 ($19000097). The first pair of bytes tells us the number of bytes that follow this hex word. Across an 80-column screen, this will be less than decimal 80 (hex $50), so the second byte is actually superfluous. It's there for future expansion, and you can use it as an indicator of a text line.

The third byte (also $00 in the example), gives the screen position of the first character; in this case it's right up against the margin. The fourth byte ($97) is used as a 'flag' plus a measure. If you translate this byte to binary (1111 1001) you will find the first binary digit is used as a Boolean indicator to tell you whether there is a carriage return at the end of the line. A 1 indicates yes and a 0 no, with the remaining 7 bits indicating how many bytes of text follow this byte.

Appleworks stores a lot of formatting information within the text storage area itself using the above two pairs of bytes as 'bookends'. This will become more apparent when we deal with larger word-processing and database files later in the series. □

Byting ProDOS Back - Part 3

IN PART 2, we jumped a bit ahead of ourselves by looking at the formatting commands in Appleworks. We haven't yet finished dealing with the way the ProDOS system allocates and controls its disk space.

Remember the primary directory is at Block 2, and ProDOS supposedly allocates available space strictly in order from the lower free to the higher block numbers. One block equals two of the old disk sectors.

Figure 1 is again the main Directory block as it appears on Pro-Byter. And to give you an overview of what's on my example disk we have Figure 2, which is the disk map produced by Pro-Byter.

Ignore the Track column on the left — we are interested only in the blocks. Probyter uses a '+' sign to signify that a block is in use, and a '.' to show that it is either empty or available for use.

You can see from this map that the first 23 blocks are in use, then 16 blocks are free, followed by a further 31 blocks in use. If ProDOS strictly allocates space in order, from the lowest to the highest block numbers, why is there gap of 16 blocks in the middle?

A glance at the directory block gives you the answer. YC.PRODOS (which is an Appleworks W/P file) has been changed and saved again, so the space previously occupied by it, is now vacant.

There have been a number of other changes that aren't apparent, but which make interpretation of the map rather confusing. For instance, the directory (Figure 1) shows that the Appleworks database file ADDRESS.FILE begins at Block $0800 (decimal 8) and is $1E (31) blocks long.

There's a curiosity here that I don't understand — but here are the facts as they relate to Appleworks files, anyway. ProDOS takes up the first seven blocks, from $00 to $06, with $06 the bit map which keeps a running record of which disk blocks are used and which are still available.

Now, the first file on the disk starts at

```
00 00 03 00
   ===========
F9 DEMO.DISK::::::: 00 0000 0000 000000
0000-0000 00 00 C3 270D 0300 0600 1B01
   ===============
2C ADDRESS.FILEEMP 19 0800 1E00 573B00
FCAC-0000 00 00 E3 7E70 FCAC-0000 0200
   ===============
1B SEEDLINGKS.TEMP 1A 1600 0100 4B0100
FCAC-0000 00 00 E3 0000 FCAC-0000 0200
   ===============
00 YC.PRODOSS.TEMP 1A 1800 1000 E01C00
FCAC-0000 00 00 E3 2000 FCAC-0000 0200
   ===============
29 YC.PRODOSS.TEMP 1A 3700 1000 CC1D00
FCAC-0000 00 00 E3 2000 FCAC-0000 0200
   ===============
00 ;;;;;;;;;;;;;;; 00 0000 0000 000000
0000-0000 00 00 00 0000 0000-0000 0000
   ===============
00 ;;;;;;;;;;;;;;; 00 00
BYTE      VALUE      A  BLOCK  PART      6
44($2C)   65($41)    A  2($02)    A       2
```

Figure 1. *The main Directory block. Note that* YC.PRODOS *has been changed and saved since Figure 1 in Part 2.*

```
                PRODOS(TM) BLOCK MAP

      TRACK          01234567 89ABCDEF
      $00-01   $0000++++++++ ++++++++
      $02-03   $0010++++++++. ........
      $04-05   $0020.......+ ++++++++
      $06-07   $0030++++++++ ++++++++
      $08-09   $0040++++++.. ........
      $0A-0B   $0050........ ........
      $0C-0D   $0060........ ........
      $0E-0F   $0070........ ........
      $10-11   $0080........ ........
      $12-13   $0090........ ........
      $14-15   $00A0........ ........
      $16-17   $00B0........ ........
      $18-19   $00C0........ ........
      $1A-1B   $00D0........ ........
      $1C-1D   $00E0........ ........
      $1E-1F   $00F0........ ........
      $20-21   $0100........ ........
      $22      $0110........

      BLOCKS USED(+):54 / FREE(.):226
```

Figure2. *The disk map produced by* Pro-Byter *for the example disk.*

$08, but from the disk map we can see that $07 is in use. What with? When I did a trace of the ADDRESS.FILE file I found that it jumps from the directory, to Block $08, then back to $07, then on to $09.

This jump forward, the back and then forward again seems to happen on all Appleworks files, but all the books say that ProDOS allocates all blocks in strict order from the bottom up.

I don't pretend to understand this apparent contradiction, but it seems to apply to all files — except seedling files, of course, which are only one block in length. It is obviously a peculiarity of Appleworks rather than ProDOS.

ADDRESS.FILE is a database file, but the same applies to all Appleworks word processing files as well. The directory pointer points to the block reserved (in the longer files) as an index for that particular file. You can see the top couple of lines on Figure 3, which is a printout of Block $08 Part A.

```
;;;;;;;;;;;;;  07090A0B 0C0D0E0F 10111213
;;'()*+,-./0  14152728 292A2B2C 2D2E2F30
12345;;;;;;;  31323334 35000000 00000000
;;;;;;;;;;;;  00000000 00000000 00000000
;;;;;;;;;;;;  00000000 00000000 00000000
;;;;;;;;;;;;  00000000 00000000 00000000
;;;;;;;;;;;;  00000000 00000000 00000000
;;;;;;;;;;;;  00000000 00000000 00000000
;;;;;;;;;;;;  00000000 00000000 00000000
;;;;;;;;;;;;  00000000 00000000 00000000
;;;;;;;;;;;;  00000000 00000000 00000000
;;;;;;;;;;;;  00000000 00000000 00000000
;;;;;;;;;;;;  00000000 00000000 00000000
;;;;;;;;;;;;  00000000 00000000 00000000
;;;;;;;;;;;;  00000000 00000000 00000000
;;;;;;;;;;;;  00000000 00000000 00000000
;;;;;;;;;;;;  00000000 00000000 00000000
;;;;;;;;;;;;  00000000 00000000 00000000
;;;;;;;;;;;;  00000000 00000000 00000000
;;;;;;;;;;;;  00000000 00000000 00000000
;;;;;;;;;;;;  00000000 00000000 00000000
;;;;;;;;;;;;  00000000 00000000 00000000
;;;;          00000000 TRACE:ADDRESS.FIL
BYTE      VALUE      c@  BLOCK  1 OF 30
44($2C)   0($00)     @   8($08)    A       2
```

Figure 3. A *printout of Block* $08 *Part* A *of the* ADDRESS.FILE.

From the left we see the file first uses Block $07, then jumps to $09 and runs progressively to Block $15. it then jumps to $27 and continues on until Block $35. The second jump was probably due to another file getting in the way at the time of recording.

The index block actually uses two sectors at a time. Part B on this disk is nothing more than a series of $00s, but there could be some $01 in here if I had a full disk, or I was using a hard disk.

The program takes the least significant byte from part A (in this case $07 ... $09 ... and so on) and the most significant byte from Part B (in this case they are all $00s).

With this Appleworks database file, the actual information didn't start until near the end of Block $0A. Both $07 and $09 were occupied with the database formatting information.

With word processing files, only one block is reserved for format information, and the data starts at the top of the next block.

As I've noted before, Block 6 (Figure 4) on a ProDOS floppy is reserved for a bit map, and for those of you who haven't come across a bit map before, I'll describe it in detail. You need to know how it works if you are going to repair the damaged files.

You can see that the sequence goes: $00 $00 $01 $FF $FE $00 $00 $00 $03 ... followed by a whole log of $FFs. These

```
::: ~::::     000001FF FE000000 03FFFFFF
              FFFFFFFF FFFFFFFF FFFFFFFF
           :  FFFFFFFF FFFFFFFF FFFFFF00
;;;;;;;;;;;;  00000000 00000000 00000000
;;;;;;;;;;;;  00000000 00000000 00000000
;;;;;;;;;;;;  00000000 00000000 00000000
;;;;;;;;;;;;  00000000 00000000 00000000
;;;;;;;;;;;;  00000000 00000000 00000000
;;;;;;;;;;;;  00000000 00000000 00000000
;;;;;;;;;;;;  00000000 00000000 00000000
;;;;;;;;;;;;  00000000 00000000 00000000
;;;;;;;;;;;;  00000000 00000000 00000000
;;;;;;;;;;;;  00000000 00000000 00000000
;;;;;;;;;;;;  00000000 00000000 00000000
;;;;;;;;;;;;  00000000 00000000 00000000
;;;;;;;;;;;;  00000000 00000000 00000000
;;;;;;;;;;;;  00000000 00000000 00000000
;;;;;;;;;;;;  00000000 00000000 00000000
;;;;;;;;;;;;  00000000 00000000 00000000
;;;;;;;;;;;;  00000000 00000000 00000000
;;;;          00000000
BYTE      VALUE      c@  BLOCK  PART      6
44($2C)   0($00)     @   6($06)    A       2
```

Figure 4. A *printout of* $06 — *the bit map* — *of the* ADDRESS.FILE. *Compare this with the disk map in Figure* 2.

$FFs are added when the disk is formatted, so they establish the length of available space on the disk. You can see that, with two parts, this block of bit map can handle a fair amount of disk space.

In binary, the hex $00 translates to eight zero bits (0000000) and is repeated twice. Hex $01 is binary 0000 0001, while $FF is eight ones, and FE seven ones and a zero.

If you space these out in groups of eight in table form, you get the following pattern:

```
00000000 00000000
00000001 11111111
11111110 00000000
00000000 00000000
00000011 ... and so on.
```

As you can see, this corresponds exactly to Pro-Byter's disk map in Figure 2, which is understandable since this is where Pro-Byter took its information from. The only oddity is that zeros are used to mark the presence of information in a block, and ones represent availability.

As the disk is over-written by changes and fresh files are added, this disk map is constantly changed. You can represent any combination of zeros and ones in an eight-bit group by choosing the right hex number.

The computer refers to this map before writing any information to a disk. The map doesn't play a part in reading from a disk — the computer has to refer to the individual file indexes since this records the sequence of blocks used — but for writing to the disk, the computer only has to find those blocks which are available, and these are any on the disk-map represented by a one. □

ATARI ROUTINES

Here are two handy Atari routines for drawing circles, and arcs and pies.

THE CIRCLE routine is quite straightforward – after having loaded the subroutine (both routines are given in the Listing), it is only necessary to specify the *x* and *y* co-ordinates of the centre point (*xl* and *yl*) and the radius (*rad*) then simply gosub Drawcircle.

The arc and pie routine is a bit more complex. Firstly, we need to tell the subroutine whether we want an arc or a pie. This is done by setting the variable 'id' (as in Australia Card) to 2 for an arc and 3 for a pie.

Next we must specify the start and finish angles. The Atari convention is that 0 degrees is horizontal to the right of the centre point and the angle increases counter-clockwise. The angles for this routine must be specified in tenths of a degree and are defined in variables 'start' and 'finish'.

All that remains is to specify the centre point and radius, then we are in business.

The example shown in Listing 2, allows you to draw a circle with a centre of 100,100 and a radius of 100, clear the screen, then draw first an arc and secondly a pie with the same centre point and radius. The angle of the arc and pie will be from 30 degrees (start = 300) to 75 degrees (finish = 750).

Software

I recently looked at two programs which attack the education market from quite different viewpoints. The first is Math Talk from First Byte Software; the second is the new favourite for kids of all ages — Donald Duck's Playground.

Talking Maths

Math Talk is very good in parts, but has a couple of faults which really need to be addressed before it could be wholeheartedly recommended.

The program has a number of modules, which allow for a variety of activities. Firstly, the parent or teacher can set up sheets of problems in the four basic areas of arithmetic — addition, subtraction, multiplication and division. The problem sheets can be printed out or called up by another part of the program.

When a sheet of problems is called up they can be attempted one by one. If a mistake is made, the talking Professor Matt A. Matic will explain how to attack the problem. But — while the explanations given are quite sound and make

```
10     fullw 2: clearw 2
20     color 1,3,1,1,1
30     x1= 100: y1 = 100: rad = 100
40     gosub DRAWCIRCLE
50     gosub DELAY
60     clearw 2
70     start = 300: finish = 750
80     id = 2: gosub ARCPIE
90     gosub DELAY
100    clearw 2
110    id = 3: gosub ARCPIE
120    gosub DELAY
130    end
140    DELAY:
150    for time = 1 to 5000: next time
160    return
170    '
50530  ' ********************************
50540  '
50550  DRAWCIRCLE
50560  poke contrl, 11
50570  poke contrl+2, 3
50580  poke contrl+6, 0
50590  poke contrl+10, 4
50600  poke contrl+12, 2
50610  poke ptsin, x1
50620  poke ptsin+2, y1
50630  poke ptsin+4, 0
50640  poke ptsin+6, 0
50650  poke ptsin+8, rad
50660  poke ptsin+10, 0
50670  vdisys(0)
50680  return
50690  '
50700  ' ********************************
50710  '
50720  ARCPIE
50730  poke contrl, 11
50740  poke contrl+2, 4
50750  poke contrl+6, 2
50760  poke contrl+10, id :
       '  primitive ID: 2 = arc; 3 = pie
50770  poke contrl+12, 2
50780  poke ptsin, x1
50790  poke ptsin+2, y1
50800  poke ptsin+4, 0
50810  poke ptsin+6, 0
50820  poke ptsin+8, 0
50830  poke ptsin+10, 0
50840  poke ptsin+12, rad
50850  poke ptsin+14,0
50860  poke intin, start
50870  poke intin+2, finish
50880  vdisys(0)
50890  return
```

Listing 2. *This example draws a circle with a centre of 100,100 and a radius of 100, clears the screen, then draws first an arc and then a pie with the same centre point and radius.*

sense to an adult, they follow a different format to that used in Australian (or at least West Australian) schools. In the lower primary grades, this inconsistency is sufficient to cause confusion and rejection by the young student.

The section of Math Talk which pits the kids against the clock in tackling 'tables' is far more successful. However, the facility for calling up a kid's score never seemed to work.

A further problem with Math Talk is the very slow reaction to mouse events. Ideally, there should be an instant reaction on screen, even if the processing for the next segment then takes place over a couple of seconds. However with Math Talk, nothing seems to happen for some time after the mouse button is pressed — you begin to wonder if the computer registered the click.

Overall, I see Math Talk as a program with great potential, but Version 1.0 is not yet a top product. If you wish to look at this program, get a primary school teacher to run through it with you before committing yourself to a purchase.

Disney Software

Donald Duck's Playground shows a refreshing approach to educational software — the kids think it's a game and learn without even knowing it. The big kids reckon it's pretty smooth too

Donald has been around on a number of machines, but the Atari version seems to be the most successful. (I imagine the Amiga could have a reasonable version, too, but we won't talk about that!)

The basic idea of the game is that Donald wishes to equip a playground for his three nephews. In order to buy the goodies, he must get a job, earn the brass, then go to the various shops and complete the transaction

There are four jobs available — train controller, packer at the fruit market, shelf stacker at the toy shop and airport luggage handler.

As a train controller, Don has to throw a series of levels to direct the train on the most direct route to pick up goods at a nominated station, then deliver them elsewhere. For each successful delivery, a small amount is added to his wages. (Note that this is all piecework and payment is only for results. The unions would have a fit!)

At the fruit market, our intrepid duck is standing in front of three large bins. As watermelons, pumpkins and lemons are hurled at him from the back of a truck. Donald has to catch them and place them in the right bin. Again it's payments for results. One complication at the fruit market is that the fruit doesn't seem to follow the laws of physics as it sails through the air — no nice parabolic curves here.

At the fruit market, our intrepid duck is standing in front of three large bins.

The toyshop requires Donald to position a ladder in front of the right part of the shelf, pick up toys from the conveyor, climb the ladder and put the toy on the shelf. This is hard. A further complication is that at frequent intervals a train will roar past, vibrating some of the stock down from the shelves. Donald must anticipate this and pull a level which places a guard in front of the toys. For every broken item, money is deducted from his wages — where's the shop steward?

At the airport, packages on a conveyor must be picked up and thrown into waiting trolleys. The packages all bear three character airport codes, as do the trolleys. The airport sequence features a nicely animated jumbo jet landing then taxiing up to the apron.

Earning your money in this game is quite hard, especially at the toyshop. The game is configured to be played using keyboard (either numeric pad or cursor keys), joystick or mouse. Quite frankly, the mouse is much too hard for this application — the joystick is best.

Having earned all this money, Donald can go over to the shops and buy up the equipment for the playground. This includes slides, climbing ropes, ladders, old boxes and so on. Having selected an item, Donald then must pay for it by placing coins in the cash register. If he doesn't have the right change he must work out how much change is due.

The fun is just beginning, as you can now walk Donald over to the park, where one of his nephews is seen sliding down slides, jumping on old boxes and climbing trees

At this point, one of the few faults with this program becomes apparent — at each change of scenery, the program goes out to disk to load the next screen. It is quite disconcerting to have young Hewie (or is it Louie?) sliding down a flying fox then have the whole thing freeze up for a few seconds while the disk drive does its thing.

Running the program from a RAM disk might overcome this problem, but I didn't have enough memory on my 520 ST for this to work. Even with a RAM disk there may be problems with the copy-protection on the master disk.

Donald Duck's Playground is a very good program indeed. It has excellent graphics, smooth action, catchy tunes and teaches money handling, arithmetic and hand-eye co-ordination. Three to six year old players will need help from mum or dad, which is the standard excuse in this household for hogging the action.

New Hardware

That West Australian innovation house, Tech Soft, has just announced its new 360 kilobyte 13 cm floppy drive for the 520 and 1040 STs. Why, I hear you ask, would anybody want to connect an old technology drive to an ST? Well it makes sense when I tell you that they have also released their IBM PC emulation program! That's right, the Atari ST can now be down-graded to run all that so-called 'industry standard' software.

Seriously, this is an immensely important innovation for people like me who work with IBM or clone computers and have Ataris at home. It is now possible to transfer data between the machines and to run IBM software such as Lotus 1-2-3, Wordstar, Microsoft Word and other well known packages on the Atari.

The execution looks a bit leisurely — I would guess it's just a bit slower than the original 4.77 MHz PC. However, for the type of use which this product is designed for, that is adequate. It's also really very impressive when you consider that the MC68000 processor in the Atari is not only carrying out all the calculations for the IBM programs, but at the same time pretending to be a totally different type of processor.

I hope to have full details of the IBM emulator package next month. In the meantime, consider that with this package, and with the current heavy discounting of Atari products, you can have an IBM and an Atari in one box for less than the price of a cheap clone!

TWENTY TURBO TIPS

Part 1

Borland International's Turbo Pascal has revolutionised the way in which many of us program. Here, Peter Hill shares twenty ways to make life with Turbo Pascal more fun!

TURBO PASCAL has more than 500,000 users worldwide. Its speed, power and *especially* its low price have done more to introduce Pascal and structured programming to the mainstream of microcomputing than all the efforts of the Pascal proponents and structured programmers combined.

Turbo Pascal has also attracted a deal of criticism. On the one hand, some of those who are familiar and comfortable with BASIC or assembly language have raised their eyebrows at the 'tyranny' of its structure and the need to pre-declare all variables; on the other, esoteric arguments about Borland's adherence to the ISO Pascal standard and potential difficulties with portability of standard Pascal programs from mainframes have been raised.

While this article is not a defence of either camp, I feel it's worth stating my own position: I use whatever language will suit my immediate needs, and for some time nearly all my needs have been met by Turbo Pascal versions 2 and 3 (although I'm looking forward with interest to the release of Borland's version of Modula 2). I don't port software between mainframes and micros; if you do, you may find the arguments about portability of some relevance. If you don't, I believe it is not unrealistic to say Turbo Pascal has now become the defacto Pascal standard for microcomputers.

I don't believe that Turbo Pascal is perfect. In particular, the limitations on program size and data space dictate against its use for major projects; it is, however, sufficiently fast and flexible for medium-size projects and writing utilities.

Having got that off my chest, it's time to turn to the meat of this article. The following Turbo tips are things I have picked up in my Turbo travels which I thought worth sharing.

Tip 1.
Disk Directories

BASIC users will initially be dismayed that the simple FILES '*.*' procedure of various BASICs is not available in Turbo Pascal. Indeed, writing such a procedure in Turbo Pascal requires some knowledge of interrupt procedures. Listing 1 (Program Diry) is suitable for incorporation within programs which require an equivalent to BASIC's FILES statement.

It is possible to modify and extend this program fragment substantially to show

```
PROGRAM Diry;

{
P.R.Hill  Last modified 27/4/1986
Copyright HILLSOFT 1986
Not for sale or commercial distribution, but may be freely
distributed
alone or within users programs
}

TYPE
   Regset       = RECORD
                    ax,bx,cx,dx,bp,si,di,ds,es,Flags : INTEGER;
                  END;
   FileNameType = ARRAY[1..80] OF CHAR;
   Str80        = STRING[80];
   Dta_Def      = RECORD
                        Filler   : ARRAY[1..21] OF BYTE;
                        Attribute: BYTE;
                        FileTime : INTEGER;
                        FileDate : INTEGER;
                        FileSize : ARRAY[1..2] OF INTEGER;
                        FileName : FileNameType;
                  END;
CONST
   Carry = 1;
   Directory = $10;
VAR
   Pattern     : STRING[40];
   Size        : REAL;
   TotalSize   :REAL;
   NrFiles     : INTEGER;
   VolLabel    : Str80;
   Marker      : CHAR;
PROCEDURE Recurse;
VAR
   Dta         : Dta_Def;
   Param       : Regset;
   SearchStr   : STRING[70];
   r1,r2       : REAL;
   DtaSave     : ARRAY[1..2] OF INTEGER;
   FUNCTION Pack_Name(VAR Location; Size : INTEGER) : Str80;
   VAR
      Counter     : INTEGER;
      InterimStr  : Str80;
      Letter      : ARRAY[1..1000] OF CHAR absolute Location;
   BEGIN
     Counter := 1;
     InterimStr := '';
     WHILE (Letter[Counter]<>chr(0)) AND (Counter <= Size) DO
     BEGIN
         InterimStr := InterimStr+Letter[Counter];
         Counter := Counter+1;
     END;
     Pack_Name := InterimStr;
   END; (Function Pack_Name)
```

Listing 1. *Listing a directory.*

```
BEGIN  (Procedure Recurse)
     WITH Param,Dta DO
     BEGIN
          ax := $2F00;
          MsDos(Param);
          DtaSave[1] := es;
          DtaSave[2] := bx;
          ax := $1A00;
          ds := Seg(Dta);
          dx := Ofs(Dta);
          MsDos(Param);
          ds : Seg(Pattern[1]);
          dx := Ofs(Pattern[1]);
          ax := $4E00;
          cx : $FF;
          MsDos(Param);
          TotalSize:=0;
          WHIL (Flags AND Carry) = 0 DO
BEGIN
   SearchStr := Pack_Name(FileName,sizeof(FileName));
   IF ((Attribute<>16)
        AND ((Attribute AND Directory) <> 0)
        AND (SearchStr <> '.')
        AND (SearchStr <> '..')) THEN
   BEGIN
        SearchStr := SearchStr+chr(0);
        ax := $3B00;
        ds := Seg(SearchStr[1]);
        dx := Ofs(SearchStr[1]);
        MsDos(Param);
        Recurse;
        ax := $3B00;
        SearchStr := '..'#0;
        ds := Seg(SearchStr[1]);
        dx := Ofs(SearchStr[1]);
        MsDos(Param);
    END
    ELSE
    BEGIN
      r1 := FileSize[1];
      r2 := FileSize[2];
      if r1 < 0 then r1 := r1+65536.0;
      if r2 < 0 then r2 := r2+65536.0;
      REPEAT SearchStr:=SearchStr+' '
           UNTIL Length(SearchStr)=13;
      Size:=(r2*65536.0+r1)/1000;
      TotalSize:=TotalSize+Size;
      IF Size<1 THEN Size:=1;
      CASE Attribute OF
        8:VolLabel:=SearchStr;
        2,35,38,39 :(this is hidden);
        16:Write(SearchStr,'sdir ',CHR(179),' ');
        ELSE
        BEGIN
          Write(SearchStr,Size:3:0,'k ',CHR(179),' ');
          NrFiles:=NrFiles+1;
                END
              END; (Case)
          END;
          ax := $4F00;
          MsDos(Param);
        END;
        ax := $1A00;
        ds := DtaSave[1];
        dx := DtaSave[2];
        MsDos(Param);
     END;
END; (Procedure Recurse)
BEGIN    (Main body of program)
      Write('Pattern ?:');
      Read(Pattern);
      IF Length(Pattern)=0 THEN Pattern:='*.*';
      ClrScr
      NrFiles:=0;
      VolLabel:='';
      Pattern := Pattern+hr(0);
      Recurse;
      WriteLn;
      HighVideo;Write('>');LowVideo;
      Write('Total of ',TotalSize:4:0,
         ' kB',' in ',NrFiles,' files   ');
      IF VolLabel<>'' THEN
      BEGIN
           HighVideo;Write('>');LowVideo;
           Write('Volume label ',VolLabel);
      END;
      WriteLn;
      Write('Press any key to continue....');
      REPEAT UNTIL KeyPressed;
      ClrScr;
END.
```

directories only, show files only, show the long file form including date and time of file creation, or to show hidden and system files. I have presented a short version because it provides the basic functions without being excessively long. In order to incorporate it in a program, you should make the following modifications:

1. change the top line from PROGRAM Diry; to PROCEDURE Diry;
2. change the last line from END. to END; {Procedure Diry}
3. place the line {$IDiry.Inc} in your program
4. name the above procedure DIRY.INC

Tip 2.
Fast Screens

Turbo Pascal generates 'well-behaved' code; that is, the code it produces does not directly address hardware, but rather works through the MS-DOS and ROM BIOS function calls. This ensures a degree of portability across various machines. However, one unfortunate consequence is functions such as writing to the screen can be *v-e-r-y* slow.

One solution to this problem is to directly address the part of memory dedicated to maintaining the screen. This approach does reduce portability, as other MS-DOS computers, such as the NEC APC III and the Hewlett-Packard HP150, use different memory addresses for the screen. But the problem is not insurmountable, since the code can easily be modified for the particular machine if the correct screen-memory address is available. If you are wondering whether this is worthwhile when compared with the simplicity of using Write or WriteLn, then I can only add this gives an almost instantaneous screen refresh for a complete screenful of text!

Listing 2 illustrates the use of the procedure Write—A—Line (WAL) instead of Write or WriteLn. The parameters which are passed to this procedure are –

Attr – the video attribute required (see Table 1);
Return – if a carriage return is required at the end of a string, pass a value of 1, else pass a value of 0;
C – the column to start the string;
R – the row to start the string;
CurrentStr – the string to display.

Essentially, this routine places the ASCII values of the characters directly into memory locations which happen to be the memory locations allocated to the video display. The value $B000 (Hexadecimal B000 or Decimal 45056) is the starting address of video memory, and C is the calculated offset from this starting address, dependent on the row and column desired. The starting address given above is for the monochrome adapter. For the colour-graphics adaptor, you should substitute $B800 for $B000 in the above. The attribute you require can be derived from the video attribute list in Table 1. You get the attributes by adding together the desired characteristics; for example –

128+2+8+64=202

will give you a bright, blinking green foreground on a red background.

A call to the procedure WAL within the body of a program might look like this:

```
ThisStr:='Bright green on black';
WAL(10,1,15,5,ThisStr);
```

The result of this will be equivalent to the standard Turbo Pascal statements:

```
ThisStr:='Bright green on black';
HighVideo;
GotoXY(15,5);
WriteLn(ThisStr);
LowVideo;
```

As you can now see, this is a more compact method of writing to the screen. As you will see when you use it, it is also *much* faster. You'll appreciate it when you have a full screen of text to present.

```
PROCEDURE WAL(Attr,Return,C,R:Integer; CurrentStr:STRING[80]);
VAR
Count,Col,LenCurrentStr  :Integer;
BEGIN
     LenCurrentStr:=Length(CurrentStr)-1;
     Col:=C+LenCurrentStr;
     C:=((R-1)*160)+((C-1)*2);
     FOR Count:=1 TO LenCurrentStr DO
     BEGIN
          Mem[$B000:C]:=Ord(CurrentStr[Count]);
          Mem[$B000:C+1]:=Attr;
          C:=C+2;
     END;
     IF Return=1 THEN GotoXY(1,R+1) ELSE GotoXY(Col,R);
END;
```

Listing 2. *Writing directly to the screen.*

Value (Decimal)	FUNCTION
128	Blink
64	Red Background
32	Green Background
16	Blue Background
8	Intensity
4	Red ForeGround
2	Green ForeGround
1	Blue ForeGround

Table 1. *Video attributes.*

Tip 3.
Redirection of Input/Output

While this process is described in the Turbo Pascal manual, it would not be unkind to say some clarification could benefit the less-technical user. MS-DOS (and, of course, PC-DOS) allows redirection of input and output. This means the output of one program can serve as the input to another, and vice-versa. Using this facility, it is very simple to write a number of small utility program which can feed off each other. In combination, these trivial programs can add up to a very flexible set of extensions to the operating system. The two examples below can be used together to create an uppercase ASCII file from a Wordstar file, in the following manner:

WS2A <Wsfilename|UCASE >ASCIIfile

In this command, the program WS2A is to take its input from the named Wordstar file and place the output of the operation

```
PROGRAM WS2A;
{$G2048,P2048}
{
ws2a....WordStar to ASCII
USEAGE
ws2a <infile >outfile
}
VAR
C :Char;

BEGIN
     REPEAT
          Read(C);
{          IF Ord(C)>127 THEN C:=Chr(Ord(C)-128);}
          C:=Chr(Ord(C) AND $7F);
          IF C<>^Z THEN Write(C);
     UNTIL C=^Z;
END.
```

Listing 3. *Converting Wordstar files to* ASCII *format.*

```
PROGRAM Ucase;
{$P2028,G2028}
{
Filter input to all upper case including WordStar type files
with the MSB set.
Useage ucase <input >output   OR
dir|ucase
}

VAR
C  :Char;

BEGIN
   REPEAT
     Read(C);
     C:=UpCase(C);
     If C<>^Z THEN Write(C);
   UNTIL C=^Z;
END.
```

Listing 4. *Lower-to-upper case conversion.*

in the named ASCII file. The '|' symbol is used to 'pipe' output from one program to another. Thus, the Wordstar file will be first converted to upper case, and the output of this procedure will be piped to the WS2A program for conversion into ASCII format.

There are a couple of points to note in the programs UCASE and WS2A: as we are using DOS 'standard' files, there is no need to specify the opening, closing or assignation of the files; and the $G and $P (get and put) compiler directives are essential, with the numbers following them indicating the buffer sizes allocated to each task. The uppercase conversion program uses the standard Turbo Pascal function UpCase.

Tip 4.
Buffered Input/Output

Turbo Pascal defines a standard type of file called 'TEXT'. This file is essentially a file of lines of text, separated by the Carriage Return/Line Feed combination. Such files are commonly found in many programming situations; for example, ASCII files created by word processors, PRN files created by Lotus 1-2-3, and Pascal source files conform to this type.

One feature of Turbo Pascal is the ability to set the buffer size of such files. The default buffer size is 128 bytes, which essentially means a block of 128 bytes is read and processed, and then a new disk access is required. Although the optimum buffer size depends on the particular application, it is certain that 128 bytes is significantly less than the optimum.

For most purposes, a buffer size of 1024 bytes will result in a faster disk-access time, with fewer and quieter disk accesses. The buffer-size parameter is set when the file variable name is declared.

For example:
```
VAR
  TextFile :TEXT[$800]
```
will give a 2048 byte (or 2 Kbyte) buffer size. Some experimentation is worthwhile here; larger buffers may give better results, but they also chew through available data storage if they are declared as global variables.

Tip 5.
Global Gobblers

Turbo Pascal has a limit of 64 Kbytes of data-storage area. This limit can be circumvented by the use of pointers, but the technique is complex and also dependent on the actual program under consideration. One way to ensure you at least get your full complement of 64 Kbytes of storage is to minimise the use of global variables, and place variables in procedures whenever possible.

Listings 5 and 6 provide examples of this. The minor difference of including the help-file specification in the Open—Help—File procedure saves 4 Kbytes of space in the data area, since the space for the help file is only allocated when it is required.

```
PROGRAM Sample1;
VAR
Input,OutPut,Help:TEXT[$1000];
PROCEDURE Working_Procedure;
BEGIN
  (* Main Program procedure*)
END;
PROCEDURE Open_Help_File;
BEGIN
     (*Open the help file*)
END;
BEGIN (Main Program)
     REPEAT
          Working_Procedure;
     UNTIL Help_Called;
     Open_Help_File;
END.
```

Listing 5. *Conserving data-storage space — 1*

Turbo Pascal generates 'well-behaved' code, which ensures a degree of portability across various machines, but which makes functions such as writing to the screen v-e-r-y slow.

```
PROGRAM Sample2;
VAR
Input,OutPut     :TEXT[$1000];
PROCEDURE Working_Procedure;
BEGIN
  (* Main Program procedure*)
END;
PROCEDURE Open_Help_File;
VAR
Help :TEXT[$1000];

BEGIN
     (*Open the help file*)
END;
BEGIN (Main Program)
     REPEAT
          Working_Procedure;
     UNTIL Help_Called;
     Open_Help_File;
END.
```

Listing 6. *Conserving data-storage space* — II.

Tip 6.
Use Libraries

After programming in Turbo Pascal for some time, you will find you are covering familiar ground. For example, opening a text file to read in data consists of a number of steps, no matter what you are going to do with that data, namely:

a) Get the file name (from input or command line parameters);
b) Check the file exists;
c) If it does, assign the name;
d) Reset the file.

If you take the time to set up appropriate libraries of tried and tested routines for these functions, that time will be rapidly paid back. Probably about 50 percent of any conventional new program will comprise routines which are either identical to, or minor modifications of, routines found in most other Turbo Pascal programs. By using library routines, you will achieve faster program development, more reliable programs due to bugs being progressively eliminated, and space savings because you will 'include' these routines in your programs.

Tip 7.
Consistent Syntax

Pascal allows the free and interchangeable use of upper and lower case letters, together with long variable names. This makes it easier to enhance program clarity. If you are also consistent in your use of case and word separators, your code will be easy to read and debugging won't be so arduous. I use a number of conventions when writing programs:

- Reserved words are all in upper case, such as WHILE.
- Variables, types and standard functions are in a combination of upper and lower case, with no separators; for example, ClrScr, CountToDate, LineBuffer[n].
- Procedures and function names are in mixed case, with the underscore as a separator, for example Count—The—Lines, Rotate—Matrix.

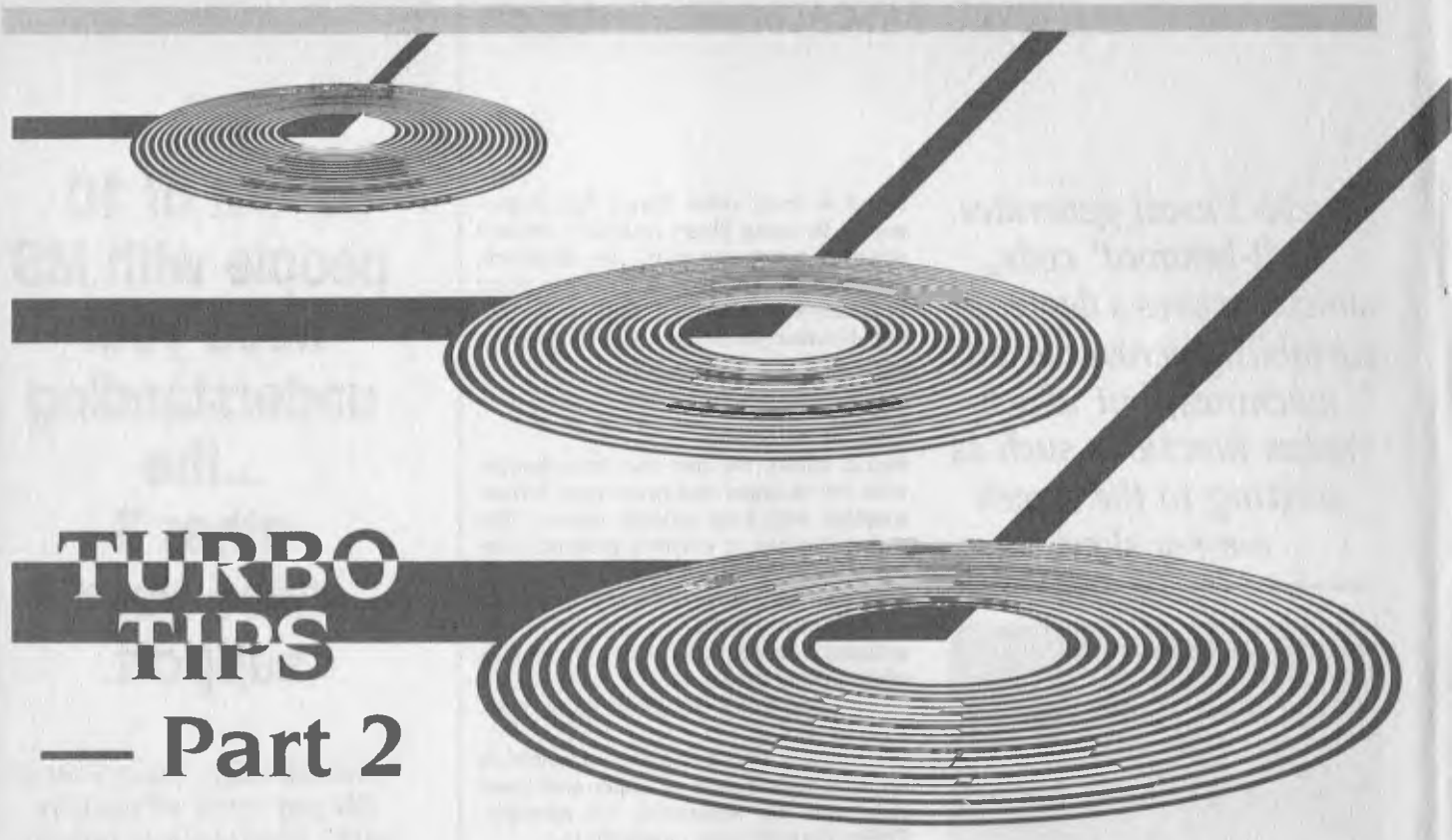
With this system, it is relatively straightforward to follow the source code. It is not important you adopt *this* convention; it is important you adopt *a* convention.

```
PROCEDURE Show_Indenting;
BEGIN
     In_One_Step;
     WHILE NOT Saturday DO
     BEGIN
        Day:=Day+1;
        WeekDay:=Succ(WeekDay);
     END;
     REPEAT
          Go_To_Work;
     UNTIL Saturday;
END; (Procedure Show_Indenting)
```

Listing 7. *Example of code indentation.*

Tip 8.
Consistent Indentation

It is similarly important to have a consistent indentation approach, and one which does not unnecessarily increase the size of the code. I have adopted an approach which indents within a block of code, which is best illustrated by example. Have a look at Listing 7: this approach allows a high degree of clarity without the overhead involved in some schemes. □



TURBO TIPS — Part 2

The speed, power and price of Turbo Pascal have given it a base of over 400,000 users who have discovered its revolutionary approach to programming. 'But, it's not perfect,' says turbo travellin' Peter Hill, and here he shares 12 more tips to make life more fun for turbo programmers.

IN PART 1 I wrote about disk directories, speeding up screen displays, redirecting input/output, buffering input/output, minimising the use of space-gobbling global variables, and the advantages of using libraries and consistent syntax and indentation. In Part 2 I've tipped the scales with the other 'half' of the info

Tip 9.
Natural Structures

The RECORD data structure provided in Turbo Pascal (and other Pascals) tends to be overlooked by newcomers from other languages, but is in fact one of the most natural constructions available in any language. Essentially, it allows an item or record to be represented as a group of features (fields or attributes). The ability to refer to this record only once and then to deal with all its features simultaneously is one of the strengths of Pascal. To give a trivial example, consider the way in which a screenful of space invaders could be addressed:

```
PROGRAM Invaders;
CONST
     Red          =1;
     Blue         =2;
     Yellow       =3;
     InvaderRows=5;
     InvaderCols=5;
TYPE
     ValueType    =1..200;
     ColourType   =Red,Blue,Yellow;
     InvaderType  =RECORD
                     Alive    :BOOLEAN;
                     Column   :INTEGER;
                     Row      :INTEGER;
                     Value    :ValueType;
                   END;
VAR
 Invaders :ARRAY[1..InvadersRows,1..InvaderCols] OF InvaderType;
```

Although setting this type of structure up takes a few lines, the beauty of being able to address each record of the array as a whole outweighs this initial effort. The short story is this where it is possible to use a natural representation of real-world (or outer space!) data, do so in preference to using fancy constructs. The results will be much more manageable when you are deeper into the program.

Tip 10.
Error Handling

With the advent of Turbo Pascal 3.0, the facility for users to write their own error-handling procedures was introduced. In

early versions of the manual, this was not included, but was referred to in the READ.ME notes on disk. The procedure is simply to:

- Assign the value of the Offset of the error to the standard variable ErrorPtr; that is, include a statement early in the main body of the code thus ErrorPtr:=Ofs(Error)
- Set up a procedure as follows:

```
PROCEDURE Error(ErrNo, ErrAddr: Integer);
BEGIN
     (error handling code)
END;
```

Unfortunately, it is essential that the last thing the error-handling procedure does is to Halt execution of the program. Any attempt to ignore the error and return to the main body of the program, or indeed any error in the error-handling routine itself, will cause Turbo Pascal to take control of the error and halt itself. This, however suggests a structure for user-written error-handlers along the following lines:

1. Give a suitable message to the user.
2. Show the Error type and Error Number for the programmer's benefit.
3. Ascertain as far as possible the nature of the problem.
4. Close files you know to be open.
5. Attempt to close files which might be open.
6. Halt.

This hierarchy tends to maximise the protection against data loss, while minimising the risk of a premature halt by Turbo Pascal.

The procedure in Figure 1 (which assumes the user has defined a file assigned to the variable InFile, and a file assigned to the variable OutFile) shows a typical error-handling procedure, which follows the above hierarchy.

Tip 11.
Plan for Overflow

The Turbo Pascal editor is limited to handling a maximum of 64 Kbytes of source code. Having written a number of large (greater than 60 Kbytes of source code) Turbo Pascal programs, I have now learned to better anticipate the problems that arise with large amounts of code.

Nothing is more pathetic than the spectacle of a programmer attempting to decide which bits of code to write out to Include files when there are only 500 bytes of space left in the editor. If there is a chance that your source code will approach 64 Kbytes, plan to develop some parts of your code as separate modules prior to the point where you are already pressed for space. The modules to be treated separately ought to be straightforward, of reasonable size and, most importantly, independent in their operation from the other modules in the main body of the program.

```
PROCEDURE Error(ErrNo, ErrAddr: Integer);
VAR
InFileProb,OutFileProb  :BOOLEAN;
BEGIN
     Clrscr;
     InFileProb:=False;
     OutFileProb:=False;
     Write(Chr(7)); (ring the bell)
     GotoXY(5,5);
     CASE Hi(ErrNo) OF
          1:Write('User Break..Bye..');
          2:BEGIN
               Write(' A fatal Error has occurred !');
               WriteLn('(Type I/O, Nr. ',Lo(ErrNo),')');
               CASE Lo(ErrNo) OF
                    2,145,153:InFileProb:=True;
                    3,242,240:OuFileProb:=True;
               END; (Case)
            END;
          3:BEGIN
             Write(' A fatal Error has occurred !');
             WriteLn('(Type Runtime, Nr. ',Lo(ErrNo),')');
          END;
     END; (Case)
    (close most probable suspect file)
     IF InFileProb THEN Close(InFile);
     IF OutFileProb THEN Close(OutFile);
     (attempt to close less probable suspect files; this
     may crash, but that is going to happen anyway!)
     IF InFileProb THEN Close(OutFile);
     IF OutFileProb THEN Close(InFile);
     Halt;
END;
```

Figure 1.

Tip 12.
Organised Back-up

Although Turbo Pascal creates .BAK files after each edit of source code, this is not a truly reliable method of back-up, since it is very easy to fall into the situation where you perform one major edit, then a small syntactical error causes the need to re-edit and save again. At this stage the old version of the structure is gone. Since Turbo Pascal is so compact, it is easy to have a disk in drive A: with Turbo Pascal and your work files, and a disk in drive B: with a copy of drive A: which is created prior to each major edit. This disk (the copy) should then be archived separately at the end of each day's work, and the previous day's disk brought out as the next day's back-up.

If this seems like a somewhat paranoid approach to back-up, believe me, it's better than the sinking feeling you get when

a) The cat eats your work disk (.BAK files and all).

b) You realise your last major edit was misdirected.

A suitable batch file on your A: drive to handle the housekeeping is SAVEIT.BAT, which reads as follows:

```
B:
COPY A:*.*
A:
DIR/W
TURBO
```

Tip 13.
Use Productivity Tools

Professional programmers in large organisations use a vast toolbox of productivity tools; typically these are expensive and powerful. For the more modest budget, there are still a number of approaches

which can be fruitful. The next three suggestions are productivity tools for the shoestring budget.

a) Keyboard Enhancers

A number of keyboard enhancers are available in the public domain. An example of these is Frank Bell's 'Newkey', which is simple, reliable and robust. Since Borland have only defined F7 and F8 on the PC's keyboard, there is a lot of scope for using Newkey to simplify coding, and also to avoid spelling errors. Pressing F1 could, for example, generate the following code –

```
WHILE       DO
    BEGIN

    END;
```

and leave the cursor between WHILE and DO.

b) ASCII *Table*

A clear,single-sheet ASCII table showing Decimal, Hex, Character and control functions visible from your keyboard is indispensable.

c) DECIMAL *to* HEXADECIMAL *Chart*

This can be very handy, especially since the Borland manual is prone to referring to such items as Error codes in Hex.

All the above are also available from Borland in more convenient form (for example, Superkey plus Sidekick).

Tip 14.
Compile to RAM

Source code can either be compiled directly to a .COM file on disk, or first to RAM and then to disk.

The second method is somewhat faster, especially for large files which may have a bug on line 2000. As soon as the program is successfully compiled to RAM, it can then be trialled in direct mode; if this is successful, it is time to compile to disk using the C option of the Compiler Options menu.

Tip 15.
Maximum Free Dynamic Memory

At the end of execution of a .COM file produced by Turbo Pascal, the program will seek to reload COMMAND.COM. If this is temporarily unavailable (for example, if it's not on the current default disk), DOS will request that a COMMAND.COM disk be inserted. This inconvenience can be avoided by setting the maximum free dynamic memory to the same value as the minimum free dynamic memory in the Compiler Options menu.

Nothing is more pathetic than the spectacle of a programmer attempting to decide which bits of code to write out to Include files when there are only 500 bytes of space left in the editor.

Tip 16.
Give Me a Break!

The default value of the compiler directive U is {$U-}. In this state, Ctrl-C will allow interruption of a program by a user only when the program is waiting for input. While developing software, it is better to set this state to {$U+}, which will allow user interruption via Ctrl-C at any time, and to accept the deterioration in speed of operation this causes. This compiler directive should be removed from the source prior to compiling for the last time, after all bugs have been eliminated.

Tip 17.
Be Liberal with Comments

Programmers who are familiar with compilers will find this an unnecessary note, but many people are more familiar with interpreters, especially BASIC, where comments actually slow execution. This is not the case in Turbo Pascal, where the compiler completely ignores any comments. Consequently, comments and notes can be sprinkled throughout the text {like this} at no cost in terms of execution time. If these comments are well chosen, they will assist you in maintaining the program or modifying it many months later.

Tip 18.
Use Both Types of Parentheses

Turbo Pascal allows comments to be surrounded by comment marks, either of { this type } or (* this type *). This raises the possibility of using each for a different purpose. Where I wish to make a comment in the source code, I do so using { this type of parentheses }. When I have suspect code, or wish to make a major change I comment it out (* thus *). Using this convention allows me to search for (* in the text, and either re-instate the commented code or eliminate it after testing.

Tip 19.
Avoid Text Searches

In longer source documents, it can take up to one minute for the Turbo Pascal editor to locate a particular string. If you are in a cycle of editing-testing-editing a section of code this, lag is unwelcome. If, however, you mark a part of your code to which you are returning frequently as the start of a block (^KB or F7), then any time you issue the command ^QB the Turbo Pascal editor will find that point immediately. This is true even if you leave the editor and compile and run a program.

Tip 20.
Read All Input as Strings

At first it may seem strange to accept the input to a real number as a string, but you only need to consider what might happen if you accept it as a real first to make sense of this tip.

The strategy is as follows (assuming we are after a positive integer) –

- Read the input as a string.
- Parse the string.
- Does it contain alpha characters?
- Does it contain a minus sign?
- Does it contain a decimal point?
- If any of the above is true, then reject, ask the user again, and repeat parsing.
- Otherwise, accept the string, and convert its type to integer.

I hope my collection of Twenty Tips above will save users of Turbo Pascal as much time as it took me to discover them; in the case of Turbo Pascal, however, the pleasure is often in the journey.

Reader's who have picked up their own tips to make life easier with Turbo Pascal are invited to share them — send them to Your Computer *and we'll pass them on to Peter Hill for comment and publication later in the year.* □

Turbo Tips

Fudging and cursing!

Two more Turbo Tips – 'Fudge' returns an integer result to multiplication by a fraction and 'Curses' allows manipulation of the cursor on a PC.

NOW THAT BORLAND, have released Turbo Pascal for the Apple MacIntosh (can the Amiga and Atari ST be far behind ?), this programming system is available for three very pervasive computer systems. The IBM PC family is supported in both MS-DOS and CP/M- 86 flavours, a very similar CP/M system is available for 8-bit users and a more-fully featured version supporting the Mac world is now making its impact.

This puts Turbo Pascal in the similar position to Microsoft Basic and some C compilers; it is becoming a lingua franca of microcomputing. On this basis, we thought the time had come to host a regular column to serve as a forum for Turbo Pascal users.

The content (and no doubt the quality) will vary from month to month; as well as tips, there will be some product information. Some content will be machine specific, whilst other parts will be more general. Importantly, I look forward to *your* participation and welcome both suggestions and queries.

Fast Fudge

To wet your appetite, Fudge returns an integer result to the multiplication of an integer number by a fractional number. Usually this would require resort to floating point operations or at least conversion to and from floating point, but you can work around that *providing* that you are confident that there will not be overflow in the first step of multiplication of the numerator of the fraction by the first integer. Unfortunately, if you aren't confident of that, Turbo Pascal won't help; there is no error checking on Integer overflow.

Amongst other things, the routine points up what could be achieved in integer arithmetic if only a *long* integer type were available in Pascal. Why bother? Timed on an IBM PC clone, the Fudge routine took 11 seconds for the 30,000 iterations, whilst the floating point calculations took 99 seconds. The answers given were –

Fudge 2733
Floating 2.7333333333E+03.

In many circumstances, the former is sufficiently close for the task.

```
PROGRAM Fudge;
VAR
i,t :INTEGER;
r   :REAL;

FUNCTION Fudge(A,B,C :INTEGER):INTEGER;
VAR
Temp     :INTEGER;
UFlow    :INTEGER;
SignFlag :INTEGER;
BEGIN
    {determine the sign of the
     result and store}
    SignFlag:=A Xor B Xor C;
    {if divide by zero return
     MaxInt; it's not infinity
     but it's certainly big}
    IF C=0 THEN Temp:=MaxInt ELSE
    BEGIN
        {we have to do the multiplication
         first or lose all accuracy}
        Temp:=Abs(A)*Abs(B);
        {if Temp is <0 then it's really a
```

```
     larger than MaxInt}
    IF Temp<0 THEN
    BEGIN
         {let's pray it's less than
          twice MaxInt and fudge again!}
         Uflow:=Maxint-Abs(Temp);
         {Treat the underflow and
          tho large part
          (MaxInt) separately}
         Temp:=Maxint;
         Temp:=Temp+(Abs(C) DIV 2);
         Temp:=Temp DIV Abs(C);
         UFlow:=UFlow+(Abs(C) DIV 2);
         UFlow:=UFlow DIV Abs(C);
         {put it back together}
         Temp:=Temp+UFlow;
    END
    ELSE
    BEGIN
         {ensure that the result is
          rounded up}
         Temp:=Temp+(Abs(C) DIV 2);
         {and do the (integer) division}
         Temp:=Temp DIV Abs(C);
    END;
END;
{correct the sign of the result}
IF SignFlag<0 THEN Fudge:=-Temp ELSE Fudge:=Temp;
END;

BEGIN  {a demonstation}
     Write(Chr(7));
     FOR I:=1 TO 3000 DO f:=Fudge(12300,2,9);
     WriteLn(f);
     Write(Chr(7));
     FOR I:=1 TO 3000 DO r:=12300*(2/9);
     WriteLn(r);
     Write(Chr(7));
END.
```

Curses!

Here we are going to look at manipulating the cursor on the IBM PC (or near offer). The routines use the ROM-BIOS services, so they will work on clones and on some other MS-DOS machines.

Turbo Pascal provides the Standard Procedure GotoXY(X,Y) to place the cursor on the screen in text mode, unlike many other Pascal and C compilers, so what more could you ask? Unfortunately, this routine is a little bit *too* well behaved; if we try to make the cursor disappear by moving it to co-ordinates off the screen, Turbo Pascal compensates for our error. Since there are many times we don't want a cursor at all, we have to resort to using a ROM-BIOS routine.

The second part of the example code shows how to change the cursor size; this is very handy to distinguish, say, insert mode in a word processor. Even more handy is to have a compiled version on disk with standard settings for some of the programmes (Reflex is an example) which don't set the cursor back to standard as they exit! Without further ado ....

```
PROGRAM Curses;
{
PlaceXY  places the cursor on  (or  off)  the screen;
Size_Curse sets the cursor to a specified number of lines;
Portability  :Limited to IBM and Clones, some MS-DOS machines;
}
TYPE
    RegPack=RECORD
                 ax,bx,cx,dx,bp,si,ds,es,flags :INTEGER;
            END;
VAR
  RecPack  :RegPack;
    IK  :CHAR;

PROCEDURE PlaceXY(X,Y :BYTE);
VAR
dh,dl,ah,al  :BYTE;
BEGIN
          {select service number two}
          ah:=2;
          al:=0;
          WITH RecPack DO
          BEGIN
               {load dh with Y and dl with X, but correct
                from  the  DOS 0,0  top-left co-ordinates to
                the  Turbo 1,1  top-left co-ordinates.}
               dh:=y-1;
               dl:=x-1;
               dx:=dh shl 8 + dl;
               {specify the page to which this applies}
               bx:=0;
               ax:=ah shl 8 +al;
               {issue  interrupt number 10 hex}
               Intr ($10,recpack);
          END;
END;
PROCEDURE  Size_Curse(StartLine,StopLine
:BYTE);
{  for the CGA,  the bottom line  is  7,
whilst for Monochrome it is 13}
VAR
ch,cl,ah,al    :BYTE;
BEGIN
          {select service number 1}
          ah:=1;
          al:=0;
          WITH RecPack DO
          BEGIN
               ch:=StartLine;
               cl:=StopLine;
               cx:=ch shl 8 + cl;
               ax:=ah shl 8 +al;
               {issue interrupt 10 e}
               Intr ($10,recpack);
          END;
END;

BEGIN
     ClrScr;
     {send cursor off-screen}
     PlaceXY(60,26);
     Read(Kbd,IK);
     {get it back}
     PlaceXY(5,5);
     {set cursor size}
     Size_Curse(1,6);
     Read(Kbd,IK);
     {restore to normal}
     Size_Curse(6,7);
     {for CGA; for MDA Size_Curse(12,13)}
     {that's all folks}
END.
```

Turbo Tips

. . . the dBase connection!

The most popular Pascal compiler is Turbo Pascal; the most popular database management system is dBase – here's how to bring the two together in a useful manner . . .

The dBase Connection

THE MOST POPULAR Pascal compiler is Turbo Pascal; the most popular database management system is dBase II/III/III Plus. Can we bring them together in a useful manner? Indeed we can, and from time to time I will present code here to allow low-level tinkering with dBase command files, databases and indexes. Of course, if you're really into dBase, you subscribe to *Your Computer* and automatically receive the free dLetter dBase newsletter, don't you?

This month we have a Turbo Pascal program which reads in dBase command files and creates an output showing the calling structure. In dBase, one command file can call another and so forth, with optional return to the original. In a large data management application it is not trivial to determine which command file is calling which, nor to guess the dependencies inherent in such a calling system. This program creates a disk file showing each command file which is called, and indenting each to show the depth of nesting of the command file. Optionally, the actual lines of source code can be sent to the disk file as well.

The program is written for the IBM-PC version of Turbo Pascal, but most of the hardware dependent features are cosmetic rather than essential, hence it is straightforward to modify the source for CP/M or MacIntosh operation (or indeed for other Pascal compilers). Modifications to produce a 'plain vanilla' CP/M version are noted in the source comments.

Since dBase II, III and III Plus command files are essentially the same, the program can equally be used on each of these systems. Sample output is as follows —

```
*Calling structure of dBase
 files commencing at:menu.prg

MENU.PRG
   PERMIN.PRG
   ACCPERM.PRG
      FINDINGR.PRG
   MODPERM.PRG
   PLCORDER.PRG
      CHKINGR.PRG
   MDDORD.PRG
   PRINMENU.PRG
      ORDLIST.PRG
         PRINT.PRG
         FPRINT.PRG
      PERMOUT.PRG
         PRINT.PRG
      HISTPROD.PRG <not found>
   DATE.PRG
Nr. of Lines processed was :2238
Nr. of Files processed was :16
```

The Turbo Pascal source to achieve this is is given in Listing 1

In the case of dBase III and dBase III Plus, the selection of the –O option not only directs all the source code to the output file, but also prepares a dBase PROCEDURE file, which can be called by a MAIN file like —

```
*MAIN.PRG
SET PROCEDURE TO OutFile
DO WHILE .T.
     DO First
ENDDO
```

In the case of complex data management systems, a PROCEDURE file can substantially increase operational speed since the necessary code is loaded into memory by the SET PROCEDURE TO command and does not have to be subsequently retrieved from disk. Whilst this can nominally be achieved by using the dBase III command editor, the limitation of this to 4K of code prevents it being performed directly except for trivial cases. Prior to taking this step, you must ensure that each module of the code is thoroughly debugged, since file size limitations might constrain your ability to subsequently edit the merged file.

This program could also be used to analyse similar calling systems with minor modifications. For example, where the dBase statement DO ProcName triggers the investigation of a called file, in R:BASE System V the equivalent commands are either INPUT ProcName or RUN ProcName IN ProcFile USING Parameters, hence the differences are syntactical rather than structural.

In the next dBase connection in the Turbo Tips series, I will present a similar utility to parse and analyse dBase command files for correct sysnpxx; dBLint, a lint type utility for dBase.

```
{$F20}  (*MS/PC-DOS only*)
PROGRAM dBTree;
{
DATE:     May '87.
BY:       Peter Hill.
PURPOSE: Show the tree structure of dBase II/III  Command files.
USEAGE:  DBTREE [MainFile] [OutPutFile] [-o[t]]
The -o option causes the output of the dBase source code to the
disk file. If this option is selected, the created file is ready
for selection as a PROCEDURE file.
The -t option causes all text to be directed to the screen
If any command line parameters are specified there is no prompting
for options; if none are specified, all are prompted.
}

CONST
MaxFiles =20;

TYPE
LongStr =STRING[255];

VAR
InFile          :ARRAY[1..MaxFiles] OF TEXT[$800];
OutF            :TEXT[$800];  (*MS/PC-DOS only; for CP/M, type is TEXT;*)
i,j,k           :INTEGER;
Depth           :INTEGER;
LineCount       :INTEGER;
FileCount       :INTEGER;
MainFile        :LongStr;
OutFile         :LongStr;
PathName        :LongStr;  (*MS/PC-DOS only*)
TextToScreen,
TextOut         :BOOLEAN;
ScrnAddr        :INTEGER;
LC              :LongStr;

FUNCTION Strip(DummyStr  :LongStr):LongStr;
{remove leading blanks}
VAR
i   :INTEGER;
BEGIN
    WHILE Copy(DummyStr,1,1)=' ' DO Delete(DummyStr,1,1);
    Strip:=DummyStr;
END;

FUNCTION Exist(VAR FileName   :LongStr):Boolean;
{determine whether a file exists}
VAR
Fil   :File;
BEGIN
     Exist:=True;
     Assign(Fil,FileName);
     {$I-}
     Reset(Fil);
     {$I+}
     Exist:=(IOResult=0);
     Close(Fil);
END;

FUNCTION IOR(VAR FileNam   :LongStr):Integer;
{determine the Input/Output result of attempting to open a file}
VAR
Fil   :File;
BEGIN
     Assign(Fil,FileName);
     {$I-}
     Reset(Fil);
     IOR:=IoResult;
     {$I+}
     Close(Fil);
END;

FUNCTION UC(DummyStr  :LongStr):LongStr;
{convert a string to UPPER CASE}
VAR
i   :INTEGER;
BEGIN
     FOR i:=1 TO Length(DummyStr) DO DummyStr[i]:=UpCase(DummyStr[i]);
     UC:=DummyStr;
END;

(* This procedure for  MS/PC-DOS only*)
PROCEDURE Get_Display;
VAR
CGAPresent :INTEGER;
BEGIN
    CGAPresent:=Mem[0:$410];
     IF ((CGAPresent AND $30)=$30) THEN ScrnAddr:=$B000 ELSE
        ScrnAddr:=$B800;
END;

   PROCEDURE Colour(Fore,Back :INTEGER);
   BEGIN
        TextColor(Fore);
        TextBackGround(Back);
   END;

(* This procedure for  MS/PC-DOS only*)
PROCEDURE WAL(Attr,C,R :Integer; CurrentStr:LongStr);
VAR
Count,Col,LenCurrentStr  :Integer;
BEGIN
     LenCurrentStr:=Length(CurrentStr);
     Col:=C+LenCurrentStr;
     C:=((R-1)*160)+((C-1)*2);
     FOR Count:=1 TO LenCurrentStr DO
     BEGIN
          Mem[ScrnAddr:C]:=Ord(CurrentStr[Count]);
          Mem[ScrnAddr:C+1]:=Attr;
          C:=C+2;
     END;
END;
FUNCTION YesNo(Dummy :LongStr):BOOLEAN;
VAR
InChar :CHAR;
BEGIN
     Write(Dummy+' (Y/N)? ');
     Colour(0,7);
     ReadLn(InChar);
     Colour(7,0);
     IF InChar IN ['Y','y'] THEN YesNo:=TRUE ELSE YesNo:=FALSE;
END;

(* This procedure for  MS/PC-DOS only*)
PROCEDURE Frame(Name :LongStr; ULX,ULY,LRX,LRY :INTEGER);
VAR
I,Attr : Integer;
BEGIN
     Attr:=112;
    WAL(Attr,ULX,ULY,Chr(201));
    WAL(Attr,ULX,LRY,Chr(200));
    WAL(Attr,LRX,LRY,Chr(188));
    WAL(Attr,LRX,ULY,Chr(187));
    FOR I := (ULX + 1) to (LRX - 1) DO
    BEGIN
        WAL(Attr,I,ULY,Chr(205));
        WAL(Attr,I,LRY,Chr(205));
    END;
    FOR I := (ULY + 1) to (LRY - 1) DO
    BEGIN
        WAL(Attr,ULX,I,Chr(186));
        WAL(Attr,LRX,I,Chr(186));
    END;
    WAL(Attr,ULX+((LRX-ULX) DIV 2)-(Length(Name) DIV 2),ULY,Name);
END;

PROCEDURE Initialise;
BEGIN
     ClrScr;
     LowVideo;
     Get_Display;                          (*MS/PC-DOS only*)
     Depth:=1;
     LineCount:=0;
     FileCount:=0;
     i:=1;
     j:=1;
     k:=1;
     PathName:='';                         (*MS/PC-DOS only*)
     TextOut:=FALSE;
     TextToScreen:=FALSE;
     WAL(112,4,1,'dBtree....by HillSoft... 1987.
          Calling structures of dBase II/III files.');
     Frame('Source',1,3,80,21);            (*MS/PC-DOS only*)
     Window(3,4,78,20);                    (*MS/PC-DOS only*)
    Frame('Status',1,23,80,25);           (*MS/PC-DOS only*)
     ClrScr;
END;
  (*Following Procedure for MS/PC-DOS only*)
PROCEDURE Get_Parms;
BEGIN
   CASE ParamCount OF
     1:BEGIN
          MainFile:=ParamStr(1);
          Write('Output File name for result ? ');
          Colour(0,7);
          ReadLn(OutFile);
          Colour(7,0);
          TextOut:=YesNo('Output all text to file ? ');
          TextToScreen:=YesNo('Output all text to screen ? ');
          IF Length(OutFile)=0 THEN Halt;
       END;
     2:BEGIN
          MainFile:=ParamStr(1);
          OutFile:=ParamStr(2);
          TextOut:=YesNo('Output all text to file?');
          TextToScreen:=YesNo('Output all text to screen?');
       END;
     3:BEGIN
          MainFile:=ParamStr(1);
          OutFile:=ParamStr(2);
          IF Pos('O',UC(ParamStr(3)))<>0 THEN TextOut:=TRUE;
          IF Pos('T',UC(ParamStr(3)))<>0 THEN TextToScreen:=TRUE;
       END;
```

Listing 1. A *Turbo Pascal program which reads in dBase command files and creates an output showing the calling structure. The program is written for the IBM-PC version of Turbo Pascal, but most of the hardware dependent features are cosmetic rather than essential, hence it is straightforward to modify the source for CP/M or MacIntosh operation (or indeed for other Pascal compilers). Modifications to produce a 'plain vanilla' CP/M version are noted in the source comments.*

```
     ELSE BEGIN
             WriteLn('USEAGE dBTree [Mainfile] [Outfile] [-o[t]]');
             WriteLn('Where -o is output of all text to Outfile');
             WriteLn('and -t is output of all text to screen.');
             Write('First (Main) file name (including extension)? ');
             Colour(0,7);
             ReadLn(MainFile);
             Colour(7,0);
             Write('Output File name for result? ');
             Colour(0,7);
             ReadLn(OutFile);
             Colour(7,0);
             TextOut:=YesNo('Output all text to file?');
             TextToScreen:=YesNo('Output all text to screen?');
             IF ((Length(OutFile)=0) OR (Length(MainFile)=0)) THEN Halt;
          END;
     END; {case}
     WAL(7,2,24,'In:');
     WAL(113,5,24,UC(MainFile));
     WAL(7,38,24,'Out:');
     WAL(113,42,24,UC(OutFile));
     WAL(7,70,24,'Line:');
END;

{a Alternative Procedure if Parameters are not supported on your
computer or version of Pascal is:

PROCEDURE Get_Parms;
BEGIN
     Write('First (Main) file to process (including extension)?');
     ReadLn(MainFile);
     Write('Output File name for result?');
     ReadLn(OutFile);
END;
*)

  (*Following Procedure for MS/PC-DOS only*)
PROCEDURE Get_Path;
BEGIN
     IF Pos('\',MainFile)<>0 THEN
     BEGIN
          PathName:=UC(MainFile);
          REPEAT
               PathName:=Copy(PathName,1,Length(PathName)-1);
          UNTIL PathName[Length(PathName)]='\';
     END;
END;

PROCEDURE Open_OutF;
BEGIN
  Assign(OutF,OutFile);
  ReWrite(OutF);
  WriteLn(OutF,'*Structure of dBase files commencing at ',
          MainFile);
  WriteLn('Structure of dBase files commencing at ',MainFile);
END;

PROCEDURE Process_File;
LABEL
L1;
VAR
CurrentLine :LongStr;
BalanceLine :LongStr;
OldX,OldY   :INTEGER;

PROCEDURE Open_New;
BEGIN
     FileCount:=FileCount+1;
     IF Exist(BalanceLine) THEN
     BEGIN
          Assign(InFile[Depth],BalanceLine);
          Reset(InFile[Depth]);
          WriteLn(OutF);
          WriteLn;
          (*Next 2 lines MS/PC-DOS only*)
          WAL(7,5,24,'                                        ');
          WAL(113,5,24,BalanceLine);
     END
     ELSE
     BEGIN
          Depth:=Depth-1;
          CASE IOR(BalanceLine) OF
               1:BEGIN
                    WriteLn(OutF,'*(not found)');
                    WriteLn(' (not found)');
                 END;
              24:BEGIN
                    WriteLn(OutF,'*(too many open files)');
                    WriteLn(' (too many open iles)');
                 END;
          END;
     END;
END;

PROCEDURE Text_Out;
BEGIN
     IF TextOut THEN
     BEGIN
          FOR k:=1 TO (3*Depth) DO Write(OutF,' ');
          WriteLn(OutF,CurrentLine);
     END;
     IF TextToScreen THEN
     BEGIN
          FOR k:=1 TO (3*Depth) DO Write(OutF,' ');
          WriteLn(CurrentLine);
     END;

END;

PROCEDURE Eliminate_Comments;
VAR
Part1     :LongStr;
Part2     :LongStr;
FoundPos  :INTEGER;

 PROCEDURE Cut_Off;
 BEGIN
      {eliminate continued comments}
      IF FoundPos<>0 THEN
      BEGIN
           IF CurrentLine[Length(CurrentLine)]=';' THEN
           REPEAT
                ReadLn(InFile[Depth],CurrentLine);
                Text_Out;
           UNTIL CurrentLine[Length(CurrentLine)]<>';'
           ELSE CurrentLine:=Copy(CurrentLine,1,FoundPos-1);
      END;
 END;

BEGIN
     CurrentLine:=Strip(CurrentLine);
     {delete blocks of text}
     IF Copy(CurrentLine,1,4)='TEXT' THEN
        REPEAT
              ReadLn(InFile[Depth],CurrentLine);
              Text_Out;
              LineCount:=LineCount+1;
              (*Next 5 lines MS/PC-DOS only*)
              IF LineCount/10 =LineCount DIV 10 THEN
              BEGIN
                   Str(LineCount:5,LC);
                   WAL(113,75,24,LC);
              END;
        UNTIL Copy(Strip(CurrentLine),1,7)='ENDTEXT';
     {get rid of comments}
     FoundPos:=Pos('*',CurrentLine);
     Cut_Off;
     FoundPos:=Pos('NOTE',CurrentLine);
     Cut_Off;
     FoundPos=Pos('&&',CurrentLine);
     Cut_Off;
     {get rid of @ 1
     FoundPos:=Pos('@',CurrentLine);
    Cut_Off;
     {get rid of ?}
     FoundPos:=Pos('?',CurrentLine);
     Cut_Off;
     {get rid of "literals"}
     FoundPos:=Pos('"'  rrentLine);
     IF FoundPos<>0 THEN
     BEGIN
        Part1:=Copy(CurrentLine,FoundPos+1,
                                Length(CurrentLine)-FoundPos);
       Curren:Line:=Copy(CurrentLine,1,FoundPos-1);
      FoundPos:=Pos('"',Part1);
       IF FoundPos<>0 THEN
         CurrentLine:=CurrentLine+
         Copy(Part1,FoundPos+1,Length(Part1)-FoundPos-1);
     END;
     FoundPos:=Pos(''',CurrentLine);


          IF FoundPos<>0 THEN
          BEGIN
             Part1:=Copy(CurrentLine,FoundPos+1,
                                    Length(CurrentLine)-FoundPos);
            CurrentLine:=Copy(CurrentLine,1,FoundPos-1);
            FoundPos:=Pos(''',Part1);
            IF FoundPos<>0 THEN
              CurrentLine:=CurrentLine+
              Copy(Part1,FoundPos+1,Length(Part1)-FoundPos-1);
          END;
      END;

PROCEDURE Main_Process;
VAR
FoundPos :INTEGER;
BEGIN
     WHILE Depth<>0 DO
```

```
      BEGIN
          WHILE NOT Eof(InFile[Depth]) DO
          BEGIN
            If Depth<1 THEN Exit;
            ReadLn(InFile[Depth],CurrentLine);
            Text_Out;
            LineCount:=LineCount+1;
            {*Next 5 lines MS/PC-DOS only*}
            IF LineCount/10 =LineCount DIV 10 THEN
            BEGIN
                Str(LineCount:5,LC);
                WAL(113,75,24,LC);
            END;
            CurrentLine:=UC(CurrentLine);
            Eliminate_Comments;
            FoundPos:=Pos('DO ',CurrentLine);
            IF ((FoundPos<>0) AND
                (Pos('ENDDO',CurrentLine)=0)) THEN
            BEGIN
               i:=FoundPos;
              j:=Length(CurrentLine)-FoundPos;
               BalanceLine:=Copy(CurrentLine,i+2,j);
               IF ((Ps(' CASE',BalanceLine)=0)
                  AND (Copy(Strip(BalanceLine),1,1)<>'&')
                  AND (Pos(' WHILE',BalanceLine)=0))
                  THEN

              BEGIN
                   BalanceLine:=Strip(BalanceLine);
                   IF Pos(' ',BalanceLine)<>0 THEN
                      BalanceLine:=Copy(BalanceLine,1,
                         Pos(' ',BalanceLine)-1);
                  IF Pos('.',BalanceLine)=0 THEN
                      BalanceLine:=BalanceLine+'.PRG';
{*MS/PC-DOS only*} IF Pos('\',BalanceLine)=0 THEN
                      BalanceLine:=PathName+BalanceLine;
                  FOR k:=1 TO (3*Depth) DO
                  BEGIN
                       Write(OutF,' ');
                       Write(' ');
                  END;
                  IF TextOut THEN Write(OutF,'*');
                  Write(OutF,BalanceLine);
                  IF TextToScreen THEN Colour(0,7);
                  Write(BalanceLine);
                  LowVideo;
                  Depth:=Depth+1;
                  Open_New;
                  Main_Process;
              END;
            END;
          END;
          IF (TextOut) AND (CurrentLine<>'RETURN') THEN
             WriteLn(OutF,'RETURN');
          Close(InFile[Depth]);
          Depth:=Depth-1;
          IF Depth<1 THEN Exit;
      END;
END;

BEGIN
     BalanceLine:=MainFile;
     Open_New;
     BalanceLine:=UC(BalanceLine);
     WriteLn(OutF,'*'+BalanceLine);
     IF TextOut THEN WriteLn(OutF,'PROCEDURE '+BalanceLine);
     IF TextToScreen THEN Colour(0,7);
     WriteLn(BalanceLine);
     LowVideo;
     Main_Process;
END;

PROCEDURE Wind_Up;
BEGIN
     WriteLn('Finished');
     WriteLn(OutF,'*Nr. of Lines processed was ',LineCount);
     WriteLn('Nr. of Lines processed was ',LineCount);
     WriteLn(OutF,'*Nr. of Files processed was ',FileCount);
     WriteLn('Nr. of Files processed was ',FileCount);
     Close(OutF);
     Window(1,1,80,25);  {*MS/PC-DOS only*}
END;
   {*The following Procedure is for MS/PC-DOS only*}
 PROCEDURE Error(Errno :ErrAdr  :INTEGER);
BEGIN
     Window(1,1,80,25);
     ClrScr;
     Write('  'Error Number:',Hi(Errno),':',Lo(ErrNo));
     WriteLn('An Irrecoverable Error Has Occurred...Sorry!');
     Close(OutF);
     Halt;
END;

BEGIN   {MAIN}
        ErrorPtr:=Ofs(Error);          {*MS/PC-DOS only*}
        Initialise;
        Get_Parms;
        Get_Pat;                       {*MS/PC-DOS only*}
        Open_OutFi;
        Process_File;
        Wind_Up;
END.
```